Obliczenia naukowe. Lista nr 1. Sprawozdanie.

Kacper Szatan nr 236478

20 October 2019

CEL

Celem listy jest zapoznanie się z językiem programowania *JULIA*, oraz z reprezentacją liczb zmiennoprzecinkowych w standardzie **IEEE 754**. Należało wykonać 7 zadań, które miały skonfrontować naszą wiedzę zdobytą na wykładzie dotyczącą reprezentacji liczb oraz sposobu wykonywania działań w komputerze.

Poniżej zajmiemy się omówieniem realizacji zadań wykonanych na pierwsze laboratorium. Przedstawimi również wyniki zaprezentowanych eksperymentów oraz wnioski jakie można wyciągnąć.

1 Zadanie 1 (Rozpoznanie arytmetyki)

1.1 Opis problemu

Wyznaczyć iteracyjnie epsilony maszynowe[1], liczby eta[2] oraz MAX dla wszystkich dostępnych typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754 (half, single, double). Otrzymane wyniki porównać z wbudowanymi funkcjami oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instalacji języka C.

```
x = Float16(1)
y = Float32(1)
z = Float64(1)
```

Listing 1.1: Zmienne dla całego zadania pierwszego. Przez podanie odpowiedniej zmiennej do zaprezentowanych funkcji, otrzymujemy wyniki dla arytmetyki zmiennopozycyjnej równej podanemu parametrowi. Pozwala to na zaoszczędzenie czasu i nie powoduje redundancji kodu.

[1] Epsilonem maszynowym macheps (ang. machine epsilon) nazywamy najmniejszą liczbę macheps > 0 taką, że $\mathrm{fl}(1.0 + \mathrm{macheps}) > 1.0$.

[2] Eta to najmniejsza taka liczba, że eta > 0.0

1.2 Rozwiązanie

```
function machEpsEx(one)
 mach_eps = one # startowo przypisujemy mach eps = 1
 checker = mach_eps
 while(one + checker > one) #
  mach_eps = checker        # checker w while == macheps / 2
  checker = checker / 2     #
  typeof(one)(checker)
  typeof(one)(mach_eps)
 end
 return mach_eps
end
```

Listing 1.2: Funkcja pozwalająca obliczyć macheps iteracyjnie. Wyznaczenie macheps polega na dzielenie potencjalnego macheps przez dwa dopóki warunek w pętli jest spełniony. Poszukiwanie liczby eta jest analogiczne dlatego nie zostało tu zaprezentowane. Główna różnica to warunek pętli while, który będzie postaci (checker > zero).

```
function maxEx(one)
 max = one
 checker = max
 while(!isinf(checker))
  max = checker
  checker = checker * 2
  typeof(one)(checker)
  typeof(one)(max)
 end
 min = etaEx(one)
 while(isinf(max * (2 - min)))
  min = min * 2
 end
 max = max * (2 - min)
 return max
end
```

Listing 1.3: Funkcja pozwalająca obliczyć MAX iteracyjnie. Polega to na wyznaczeniu maksymalnej liczby niebędącej nieskończonością, którą otrzymujemy w dwóch krokach. Na początku mnożymy jedynkę tylę razy przez 2 aż kolejne wymnożenie da nam nieskończoność. Następnie wyliczoną liczbę mnożymy razy (2 - minimum). Jeśli wynik wciąż jest nieskończonością zwiększamy minimum dwukrotnie i powtarzamy drugi krok. Zaczynamy od miniumum = eta.

1.3 Wyniki

Precyzja	macheps	eps()	float.h (macheps)	eta	nextfloat(0.0)	floatmin()
Float16	0.000977	0.000977	-	6.0e-8	6.0e-8	6.104e-5
Float32	1.192093e-7	1.192093e-7	1.192093e-7F	1.0e-45	1.0e-45	1.175e-38
Float64	2.220446e-16	2.220446e-16	2.220446e-16L	5.0e-324	5.0e-324	2.225e-308

Table 1: Wyniki iteracyjnie wyliczonego macheps, oraz liczby eta. Zestawione z danymi z pliku nagłówkowego *float.h* oraz funkcjami wbudowanymi. Informacyjnie: zamiast funkcji nextfloat(0.0), można wykorzystać funkcję eps() z argumentem 0.0 .

Precyzja	maxEx()	floatmax()	float.h
Float16	6.55e4	6.55e4	-
Float32	3.4028235e38	3.4028235e38	3.4028235e+38F
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e+308

Table 2: Wyniki iteracyjnie wyliczonego MAX. Zestawione z danymi z pliku nagłówkowego *float.h* oraz funkcjami wbudowanymi

1.4 Wnioski

Metoda iteracyjna poprawnie wyznacza macheps. To znaczy, że wyniki są zgodne z wywołaniami funkcji wbudowanych. Ponadto są również zgodne z odczytem z pliku nagłówkowego *float.h*. Precyzja arytmetyki określana przez nas jako ϵ to właściwie macheps podzielony przez dwa.

Wyznaczenie liczby eta również się powiodło. Jednakże jest ona mniejsza niż wywołanie funkcji floatmin(). Jest to spowodowane tym, że funkcja floatmin() zwraca liczbę, którą możemy określić jako minimum znormalizowane (MIN_{nor}). Za to liczba eta, jak i wywołanie funkcji nextfloat(0.0) zwraca nam minimum subnormalne (MIN_{sub}). Różnica w zapisie tych liczb została przedstawiona za pomocą funkcji bitstiring().

$$MIN_{nor} = 2^{C_{min}} = 0|0000000000|100000...0000$$

$$MIN_{sub} = 2^{-(t-1)}2^{C_{min}} = 0|0000000000|000000...0001$$

Trzecia część zadania polegała na iteracyjnym wyznaczeniu MAX. Skorzystano tu z wbudowanej funkcji *isinf(liczba)*, sprawdzającej czy podana liczba jest nieskończonością. Na początku obliczamy maksymalną liczbę, która jest postaci $x = 2^k k \epsilon \mathcal{Z}$. Następnie aby obliczyć właściwe maksimum szukamy liczby postaci $MAX = x*(2-m)$. Gdzie m jest najmniejszą liczbą przy której zaprezentowane wyrażenie nie jest nieskończonością.

2 Zadanie 2 (Twierdzenie Kahan'a)

2.1 Opis problemu

Kahan stwierdził, że epsilon maszynowy (macheps) można otrzymać obliczając wyrażenie $3(4/3-1)-1$ w arytmetyce zmiennopozycyjnej. W tym zadaniu eksperymentalnie sprawdzamy poprawność tego twierdzenia, dla wszystkich typów reprezentacji (half, float, double).

2.2 Realizacja

```
function machEpsKahan(one) # Kahan formula 3(4/3-1)-1
 tmp1 = typeof(one)(typeof(one)(4) / typeof(one)(3)) # 4 / 3 = tmp1
 tmp2 = typeof(one)(tmp1 - typeof(one)(1)) # tmp1 - 1 = tmp2
 tmp3 = typeof(one)(typeof(one)(3) * tmp2) # 3 * tmp2 = tmp3
 mach_eps = typeof(one)(tmp3 - typeof(one)(1)) # tmp3 - 1 = mach_eps
 return mach_eps
end
```

Listing 2.1: Funkcja sprawdzająca poprawność twierdzenia Kahan'a. Parametry podawane do funkcji znajdują się na Listingu 1.1

2.3 Wyniki

Precyzja	macheps	Kahan
Float16	0.000977	-0.000977
Float32	1.192093e-7	1.192093e-7
Float64	2.220446e-16	-2.220446e-16

Table 3: Wyniki wyliczeń macheps metodą Kahan'a. Zestawione z faktycznym epsilonem maszynowym.

2.4 Wnioski

Wyniki są równe co do wartości bezwzględnej. W precyzji Float16 i Float64 otrzymujemy wynik o przeciwnym znaku, a we Float32 dostajemy dokładny wynik.

3 Zadanie 3

3.1 Opis problemu

W zadaniu trzecim mamy do zbadania rozmieszczenie liczb w artmetyce Float64 w zadanych przedziałach [0.5 , 1], [1 , 2], [2 , 4]. W zadaniu podano, że krok z jakim rozmieszczone są liczby na przedziale [1 , 2] to $\delta = 2^{(-52)}$

3.2 Realizacja

W zadanych przedziałach wyliczano deltę z następującego równania $\delta = nextfloat(r) - r$ gdzie r ϵ do przedziału który aktualnie badamy. Jeżeli po dodaniu do naszej liczby delty nie otrzymujemy kolejnej liczby to znaczy, że 'odległość' między liczbami się zwiekszyła i na nowo trzeba wyliczyć deltę. Badając punkty stykowe między przedziałami udało się wyznaczyć 'krok' dla każdego z przedziałów. Porównano na końcu wartości delt oraz ich reprezentacje bitowe.

3.3 Wyniki

Przedział	δ	bitstring(δ)
(0.5 , 1)	$2^{-53} \approx 1.1102230246251565e-16$	0|0111100101|000000...000000
(1 , 2)	$2^{-52} \approx 2.220446049250313e-16$	0|0111100101|100000...000000
(2 , 4)	$2^{-51} \approx 4.440892098500626e-16$	0|0111100110|000000...000000

Table 4: Wyliczenia rozmieszczenia liczb na zadanych przedziałach.

3.4 Wnioski

Po porównaniu wartość oraz reprezentacji bitowych δ możemy zauważyć pewną zależność. Im liczby są bardziej oddalone od zera tym odległość miedzy nimi staje się większa. Jednocześnie krok na przedziale $[2^k, 2^{k+1}] = 2^{1-t} * 2^k$

4 Zadanie 4

4.1 Opis problemu

W arytmetyce Float64 należało eksperymentalnie wyznaczyć liczbę zmiennopozycyjną $x\epsilon(1,2)$ taką, że $x * (\frac{1}{x}) \neq 1$. Następnie znaleźć najmniejszą taką liczbę.

4.2 Realizacja

```
x = Float64(1)
min = -floatmax(Float64)
function find(x)
 while (x < Float64(2))
  if (Float64(x * (Float64(1 / x))) != Float64(1))
   println(x)
   break
  end
 x = nextfloat(x)
 end
end
```

Listing 4.1: Funkcja pozwalająca obliczyć liczbę spełniająca warunki zadania mniejszą od dwóch i większą lub równą od podanego parametru. W pętli poszukujemy kolejnych liczb metodą nextfloat() dopóki nie znajdziemy takiej, która spełnia nasz warunek.

4.3 Wyniki

Parametr	x
Float64(1)	1.000000057228997
-floatmax(Float64)	-1.7976931348623157e308

Table 5: Wyniki otrzymane z wywołania funkcji.

4.4 Wnioski

Początkowa implementacja funkcji wypisywała każdą liczbę, która spełnia warunek w zadanym przedziale. Wyjście programu zostało 'zalane' przez takie liczby. Może dziwić, że nawet na, mogłoby się wydawać, niewielkim przedziale jak (1, 2), można znaleźć aż tak wiele takich liczb. Pokazuje nam to jak zwykłe proste operacje mnożenia i dzielenia potrafią przekłamać faktyczny wynik.

5 Zadanie 5

5.1 Opis problemu

W zadaniu piątym należy zaimplementować cztery algorytmy sumowania dla podanych wektorów danych:

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

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

- "w przód" $\sum_{i=1}^{n=5} x_i y_i$.
- "w tył" $\sum_{i=n=5}^{1} x_i y_i$.
- od największego do najmniejszego (dodaj dodatnie liczby w porządku od największego do najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego, a następnie daj do siebie obliczone sumy częściowe).
- od najmniejszego do największego (przeciwnie do metody (c)).

5.2 Realizacja

```
function sumA(x, y, n)
 s = typeof(x[1])(0)
 for i = 1: n
  s = s + typeof(x[1])(x[i] * y[i])
  s = typeof(x[1])(s)
 end
 return s
end
```

Listing 5.1: Funkcja obliczająca sumę w przód (podpunkt pierwszy). Pominięto implementację drugiego podpunktu, gdyż jest analogiczny.

```
function sumD(x, y, n)
 t = sort(newTab(x, y, n))
 sum_positive = typeof(x[1])(0)
 sum_negative = typeof(x[1])(0)
 for i = 1 : n
  if i > 2 # DWIE PIWERWSZE LICZBY SA UJEMNE
   for j = i - 1 :-1: 1
    sum_negative = typeof(x[1])(sum_negative + t[j])
   end
   for k = i : n
    sum_positive = typeof(x[1])(sum_positive + t[k])
   end
   break
  end
 end
 return typeof(x[1])(sum_negative + sum_positive)
end
```

Listing 5.2: Funkcja pozwalająca obliczyć sumę zaczynając od najbliższych zera wartości (podpunkt ostatni). Obliczono sumy częściowe ujemną, dodatnią oddzielnie i na końcu je dodano. Pominięto podpunkt trzeci, gdyż jest bardzo podobny w realizacji do podpunktu czwartego.

5.3 Wyniki

Precyzja	Algorytm 1	Algorytm 2	Algorytm 3	Algorytm 4
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	1.0251881368296672e-10	-1.5643308870494366e-10	0.0	0.0
Prawidłowa wartość	-1.00657107000000e-11			

Table 6: Wynik sumowania przy zastosowaniu różnych algorytmów (różnej kolejności dodawania).

5.4 Wnioski

Od razu widać, że wyniki w arytmetyce Float64 są bliższe faktycznemu wynikowi sumowania. Najbliżej osiągnięcia poprawnego wyniku był algorytm numer 2. Należy zatem odpowiednio dobierać algorytm do danych oraz operacji jakie checmy wykonać. Niestety żaden z algorytmów nie był w stanie wyliczyć poprawnie wyniku.

6 Zadanie 6

6.1 Opis problemu

Zadanie szóste polega na policzeniu w arytmetyce Float64 wartości następujących funckji:

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

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

, dla $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$ Chociaż $f = g$ w sensie matematycznym to należy sprawdzić, która funkcja daje bardziej wiarygodne wyniki.

6.2 Realizacja

```
function f(x)
 return Float64(Float64(sqrt(Float64(square(Float64(x)))+1.0))-1.0)
end
function g(x)
 return Float64(Float64(square(Float64(x))) /
 Float64((Float64(sqrt(Float64(square(Float64(x)))+1.0))+1.0)))
end
```

Listing 6.1: Implementacja funkcji f i g.

6.3 Wyniki

Parametr (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
8^{-16}	0.0	6.310887241768095e-30
8^{-17}	0.0	9.860761315262648e-32
8^{-18}	0.0	1.5407439555097887e-33
8^{-19}	0.0	2.407412430484045e-35
8^{-20}	0.0	3.76158192263132e-37

Table 7: Wyniki funkcji f i g dla pierwszych 20 argumentów.

6.4 Wnioski

Funkcja f jest mniej dokładna niż funkcja g. Jest to spowodowane odejmowaniem liczb 'leżących' blisko siebie (w funkcji f), co powoduje utratę cyfr znaczących. W efekcie funkcja f już dla $x = 8^{-9}$ w wyniku daje nam 0.0.

7 Zadanie 7

7.1 Opis problemu

W ostatnim zadaniu należało obliczyć przybliżoną wartość pochodnej $f(x)$ w punkcie x ze wzoru:

$$f_0'(x_0) \approx \tilde{f}_0'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

Dla funkcji $f(x) = sin(x) + cos(3x)$ w punkcie $x_0 = 1$.
Obliczyć także błąd $|f_0'(x_0) - \tilde{f}_0'(x_0)|$. Aby to zrealizować wyznaczono dokładnie pochodną funkcji f która wynosi $f'(x) = cos(x) - 3sin(3x)$.
Przeprowadzić badanie dla $h = 2^{-n}$ $(n = 0, 1, 2, 3, ...54)$.

7.2 Realizacja

```
function f(x)
 return Float64(sin(x) + cos(3.0 * x))
end
```

Listing 7.1: Funkcja f.

```
function derivativeReal(x)
 return Float64(cos(x) - 3.0 * sin(3.0 * x))
end
```

Listing 7.2: Dokładna pochodna funkcji $f = f'$.

```
function derivativeAprox(x,h)
 return Float64((Float64(f(Float64(x+h)) - f(x))) / Float64(h))
end
```

Listing 7.3: Przybliżona pochodna funkcji $f = \tilde{f}'$.

```
function error()
 for n = 0:54
  h = Float64(2^(Float64(-n)))
  println("Error for h=2^(-", n, ") = ",
  abs(derivativeReal(1) - derivativeAprox(1, h)))
  println("1 + h = ", Float64(1 + h))
 end
end
```

Listing 7.4: Funkcja error() potrafi obliczyć błąd jaki został popełniony podczas przybliżenia oraz wynik $1 + h$ dla kolejnych wartości h.

7.3 Wyniki

Parametr (h)	$\tilde{f}'(1)$	$\lvert f_0'(1) - \tilde{f}_0'(1)\rvert$	$1+h$
2^{-0}	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	1.8704413979316472	1.753499116243109	1.5
2^{-2}	1.1077870952342974	0.9908448135457593	1.25
2^{-3}	0.6232412792975817	0.5062989976090435	1.125
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-15}	0.11706539714577957	0.00012311545724141837	1.000030517578125
2^{-16}	0.11700383928837255	6.155759983439424e-5	1.0000152587890625
2^{-17}	0.11697306045971345	3.077877117529937e-5	1.0000076293945312
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-27}	0.11694231629371643	3.460517827846843e-8	1.0000000074505806
2^{-28}	0.11694228649139404	4.802855890773117e-9	1.0000000037252903
2^{-29}	0.11694222688674927	5.480178888461751e-8	1.0000000018626451
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-37}	0.1169281005859375	1.4181102600652196e-5	1.000000000007276
2^{-38}	0.116943359375	1.0776864618478044e-6	1.000000000003638
2^{-39}	0.11688232421875	5.9957469788152196e-5	1.000000000001819
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.11694228168853815	1.0
2^{-54}	0.0	0.11694228168853815	1.0

Table 8: Wyniki pochodnej przybliżonej, popełnionego błędu względnego oraz wyrażenia $1+h$ dla poszczególnych wartości parametru h.

7.4 Wnioski

Początkowo zmniejszanie parametru h daje pożądane efekty gdyż przybliżenia pochodnej funkcji f stają się coraz bardziej dokładne. Jednakże dzieje się tak tylko do wartości $h = 2^{-28}$. Po dalszym zmniejszaniu parametru h dostajemy coraz większy błąd. Jest to zapewne spowodowane odejmowaniem liczb bliskich siebie $f(x_0+h) - f(x_0)$. W pewnym momencie h staje się tak małe, że zostaje pochłonięte przez $x_0 = 1$, co dobrze ilustrują wyniki ostatniej kolumny w tabelce ósmej.