Obliczenia naukowe lista nr 1

Sebastian Woźniak 268491

October 2023

1 Rozpoznanie arytmetyki - *macheps*, *eta*, *MAX*

Programy z zadania pierwszego mają na celu obliczenie epsilonów maszynowych, liczby maszynowe eta oraz maksymalną liczbę dla wszystkich typów zmnienno-przecinkowych.

1.1 Epsilon maszynowy

Jest to najmniejsza liczba ϵ w danej arytmetyce taka, że $1.0 + \epsilon > 1.0$. Aby obliczyć epsilon maszynowy inicjalizujemy zmienną *epsilon* jako 1.0 i dzielimy ją przez 2 tak długo jak spełnione jest równanie $1.0 + epsilon > 1.0$. Algorytm prezentuje się następująco:

```
epsilon = 1.0
while 1.0 + (epsilon / 2.0) > 1.0
     epsilon /= 2.0
end
```

Otrzymane wyniki oraz ich porówanie z wynikiem zwracanym przez funkcję eps(T):

```
Obliczony epsilon: 0.000977
Wartość z eps(Float16): 0.000977

Obliczony epsilon: 1.1920929e-7
Wartość z eps(Float32): 1.1920929e-7

Obliczony epsilon: 2.220446049250313e-16
Wartość z eps(Float64): 2.220446049250313e-16
```

W pliku nagłówkowym **float.h** w języku C możemy znaleźć zdefiniowane wartości epsilonów dla typów **float**, **double** oraz **long double**:

```
FLT_EPSILON     1.19209290E-07
DBL_EPSILON     2.2204460492503131E-16
LDBL_EPSILON    2.2204460492503131E-16
```

Jak widać **obliczone epsilony pokrywają się z ich wartością zawracaną przez funkcję eps oraz tą zapisaną w pliku nagłówkowym float.h**.

Jaki związek ma liczba macheps z precyzją arytmetyki (oznaczaną na wykładzie przez ϵ)?

W celu odpowiedzenia na to pytanie nałęży przypomnieć jak zdefiniowana była precyzja arytmetyki. Była to liczba $\epsilon = \frac{1}{2}\beta^{1-t}$, gdzie β to baza rozwinięcia, natomiast t to liczba bitów mantysy.
We wszystkich naszych przypadkach bazą rozwinięcia będzie liczba 2. Liczba cyfr w mantysie dla Float16 to 10, dla Float32 - 23, a dla Float64 - 52. Obliczmy, więc precyzję arytmetyki dla tych typów:

```
Float16: 1/2 * 2^1 * 2^-10 = 2^-10 = 0.000977
Float32: 1/2 * 2^1 * 2^-23 = 2^-23 = 1.19209290E-07
Float64: 1/2 * 2^1 * 2^-52 = 2^-52 = 2.2204460492503131E-16
```

Z otrzymanych wyników widać, że **epsilon maszynowy dla danej arytmetyki jest równy jej precyzji**.

1.2 Liczba maszynowa *eta*

Liczbę *eta* definujemy jako najmniejszą liczbę, która spełnia nierówność $eta > 0.0$. W celu wyliczenia jej wartości możemy skorzystać z tego samego algorytmu jak dla epsilonów maszynowych z lekko zmodyfikowanym warunkiem pętli:

```
epsilon = 1.0
while epsilon / 2.0 > 0.0
     epsilon /= 2.0
end
```

Porównajmy otrzymane wyniki z wartościami zwracanymi przez funkcje nextfloat(Float16(0.0)) itd.:

```
Obliczony Float16 Machine Eta: 6.0e-8
nextfloat(Float16(0.0)): 6.0e-8

Obliczony Float32 Machine Eta: 1.0e-45
nextfloat(Float32(0.0)): 1.0e-45

Obliczony Float64 Machine Eta: 5.0e-324
nextfloat(Float64(0.0)): 5.0e-324
```

Jaki związek ma liczba eta z liczbą MIN_{sub}? Liczbę MIN_{sub} obliczamy ze wzoru:

$$MIN_{sub} = 2^{1-t} * 2^{c_{min}}$$

Wartość t to liczba cyfr mantysy z przedziału $[1,2)$, natomiast c_{min} to minimalna cecha. Wzór na c_{min} zapisujemy:

$$c_{min} = -2^{d-1} + 2$$

Gdzie d jest liczbą cyfr użytych do zapisania cechy. Rozpiszmy dane dla poszczególnych arytmetyk i obliczmy MIN_{sub}:

```
Float16:
d = 5, t = 10
```

$MIN_{sub} = 2^{-10} * 2^{-2^4+2} = 2^{-24} = 6.0e - 8$

```
Float32:
d = 8, t = 23
```

$MIN_{sub} = 2^{-23} * 2^{-2^7+2} = 2^{-149} = 1.0e - 45$

```
Float64:
d = 11, t = 52
```

$MIN_{sub} = 2^{-52} * 2^{-2^10+2} = 2^{-1074} = 5.0e - 324$

Jak widać **najmniejsza dla arytmetyki liczba w postaci nieznormalizowanej jest równa liczbie maszynowej eta**.

Co zwracają funkcje floatmin(Float32) i floatmin(Float64) i jaki jest związek zwracanych wartości z liczbą MIN_{nor}?

Liczba MIN_{nor} oznacza najmniejszą możliwą do zapisania liczbą w postaci znormalizowanej. Korzystając z poprzednio obliczonych c_{min} możemy wyprowadzić MIN_{nor} ze wzoru:

$$MIN_{nor} = 2^{c_{min}}$$

Porównajmy wartość zwracaną przez funkcję floatmin do wartości MIN_{nor} wyliczonej ze wzoru dla Float32 oraz Float64:

```
Wartość zwrócona przez floatmin(Float32): 1.1754944e-38
MINnor = 2^-126 = 1.1754944e-38

Wartość zwrócona przez floatmin(Float64): 2.2250738585072014e-308
MINnor = 2^-1022 = 2.2250738585072014e-308
```

Wartość zwracana przez funkcje floatmin pokrywa się z wartością MIN_{nor} dla danej arytmetyki.

1.3 Liczba MAX

Celem jest uzyskanie największej liczby, którą da się zapisać używając podanej do funkcji arytmetyki. Obliczenia zostały wykonane poniższym algorytmem:

```
max_value = nextfloat(zero(T))  # MAX inicjalizowany jako eta
# Powiększaj wartość dwukrotnie dopóki następne działani nie sprawi
# że max_value = INF
while !isinf(max_value * convert(T, 2.0))
     max_value *= convert(T, 2.0)
end

rest = T(max_value) # reszta brakująca do maksymalnej wartości
# Zmniejszaj 'rest' dopóki nie stanie się liczbą maszynową eta
while rest + max_value > max_value
     # Dodawaj 'rest' do max_value dopóki max_value != INF
    while !isinf(max_value + rest)
        max_value += rest
    end
    rest /= convert(T, 2.0)
end
```

Poniżej znajdują się otrzymane wyniki porównane do wartości zwracanych przez funkcję floatmax:

```
Obliczony MAX dla Float16: 6.55e4
floatmax(Float16): 6.55e4

Obliczony MAX dla Float32: 3.4028235e38
floatmax(Float32): 3.4028235e38

Obliczony MAX dla Float64: 1.7976931348623157e308
floatmax(Float64): 1.7976931348623157e308
```

Otrzymane wyniki się pokrywają. Porównajmy je jeszcze do maksymalnych wartośći znajdujących się w pliku nagłówkowym float.h w języku C:

```
FLT_MAX = 3.402823e+38
DBL_MAX = 1.79769e+308
LDBL_MAX = 1.7976931348623158e+308
```

FLT_MAX oraz DBL_MAX pokrywają sięz obliczonym MAX dla kolejno dla Float32 oraz Float64. Brakuje odpowiednika dla Float16.

1.4 Podsumowanie

Maksymalna możliwa do zapisu liczba rośnie, a minimalna dodatnia liczba maleje tak samo jak epsilon maszynowy wraz z rosnącą liczbą bitów użytych do zapisu liczby w standardzie IEEE-754.

2 Wzór Kahana

Według Kahana wzór $3(4/3-1)-1$ pozwala na obliczenie epsilonu maszynowego w arytmetyce zmiennoprzecinkowej. Porównajmy wynik uzyskany z tego wzoru z wartością funkcji eps:

```
Float16
3 * ( 4/3 - 1) - 1 = -0.000977
Wartość z eps(Float16): 0.000977

Float32
3 * ( 4/3 - 1) - 1 = 1.1920929e-7
Wartość z eps(Float32): 1.1920929e-7

Float64
3 * ( 4/3 - 1) - 1 = -2.220446049250313e-16
Wartość z eps(Float64): 2.220446049250313e-16
```

Wartość bezwzględna otrzymanych wyników pokrywa się z rzeczywistą wartością epsilonu maszynowego.

3 Równomierne rozmieszczenie liczb zmiennopozycyjnych

Celem tego programu jest obliczenie czy liczby są równomiernie rozmieszczone w sprawdzanym przedziale innymi słowy (dla przedziału $[1, 2]$) czy można zapisać je jako:

$$x = 1 + k * \delta, k = 1, 2, ..., \delta^{-1} - 1$$

Ze względu na to, że sprawdzenie całego przedziału jest bardzo kosztowne jesteśmy w stanie sprawdzić tylko jego części. Algorytm dla podanej delty przechodzi przez 1000 kolejnych liczb i sprawdza czy odległość między każdą z nich jest równa δ. Jeżeli pętla dojdzie do końca bez wykrycia błędu wyświetli się poniższy komunikat:

```
Sukces. Rozmieszczenie jest równomiernie z krokiem delta.
```

Z wyników po sprawdzeniu 1000 pierwszych wartości w przedziale oraz 1000 ostatnich możemy wywnioskować, że są one równomiernie rozmieszczone z krokiem 2.220446049250313e-16.
Nie zmieniając delty sprawdźmy przedział $[\frac{1}{2}, 1]$ oraz $[2, 4]$:

```
[1/2, 1]
[Nierównomierne rozmieszczenie]
liczba 0.5000000000000001 nie jest możliwa do zapisania jako 1 + k*2^-52
Wykryta delta: 1.1102230246251565e-16
```

```
[2, 4]
[Nierównomierne rozmieszczenie]
liczba 2.0000000000000004 nie jest możliwa do zapisania jako 1 + k*2^-52
Wykryta delta: 4.440892098500626e-16
```

Widzimy, że dla [2, 4] $\delta = 2^{-51}$ a dla $[\frac{1}{2}, 1]$ $\delta = 2^{-53}$.
Liczby w przedziałach pomiędzy kolejnymi potęgami 2 są równomiernie rozmieszczone jednak przedziały te posiadają tyle samo liczb co sprawia, że odległości między nimi są coraz większe.

4 Liczba spełniająca nierówność $x * \frac{1}{x}$!= 1

Algorytm:

```
while num * (one(Float64)/num) == one(Float64)
    num = nextfloat(num)    # sprawdź następną liczbe
end
```

Wynik:

```
Najmniejszą liczbą w przedziale (1.0, 2.0) spełniająca warunek:
x * 1/x != 1 jest 1.000000057228997
```

Wniosek:
Ograniczenia standardu IEEE-754 mogą prowadzić do otrzymywania niepoprowanych wyników działań. Należy brać to pod uwagę operując na liczbach zmiennoprzecinkowych w algorytmach.

5 Iloczyn skalarny

W tym programie obliczany jest iloczyn skalarny dwóch wektorów z liczbami zmiennoprzecinkowymi na różne sposoby.

- "w przód"

  ```
  S = T(0.0) # suma
  for i in 1:n
      S += T(x[i])*T(y[i])
  end
  ```

- "w tył"

  ```
  S = T(0.0)  # suma
  for i in 1:n
      S += T(x[n-i+1])*T(y[n-i+1])
  end
  ```

- liczby dodatnie malejąco, ujemne rosnąco

```
products = []   # pusta tablica na iloczyny
for i in 1:n
    push!(products, T(x[i])*T(y[i]))
end
# posortowana malejąco tablica liczb nieujemnych
positive_nums = sort(filter(a -> a >= 0, products), rev=true)
# posortowana rosnąco tablica liczb ujemnych
negative_nums = sort(filter(a -> a < 0, products))
S_1 = T(0.0) # suma nieujemnych liczb
for i in positive_nums
    S_1 += T(i)
end
S_2 = T(0.0) # suma ujemnych liczb
for i in negative_nums
    S_2 += T(i)
end
S = T(S_1) + T(S_2)
```

- liczby dodatnie rosnąco, ujemne malejąco

```
products = []   # pusta tablica na iloczyny
for i in 1:n
    push!(products, T(x[i])*T(y[i]))
end
# posortowana malejąco tablica liczb nieujemnych
positive_nums = sort(filter(a -> a >= 0, products))
# posortowana rosnąco tablica liczb ujemnych
negative_nums = sort(filter(a -> a < 0, products), rev=true)
S_1 = T(0.0) # suma nieujemnych liczb
for i in positive_nums
    S_1 += T(i)
end
S_2 = T(0.0) # suma ujemnych liczb
for i in negative_nums
    S_2 += T(i)
end
S = T(S_1) + T(S_2)
```

Oto otrzymane wyniki w precyzjach Float32 i Float64:

```
[Pojedyńcza precyzja - Float32]:
Iloczyn 'w przód':      -0.4999443
```

```
Iloczyn 'w tył':        -0.4543457
Iloczyn malejąco:       -0.5
Iloczyn rosnąco:        -0.5

[Podwójna precyzja - Float64]:
Iloczyn 'w przód':      1.0251881368296672e-10
Iloczyn 'w tył':        -1.5643308870494366e-10
Iloczyn malejąco:       0.0
Iloczyn rosnąco:        0.0
```

Prawdziwą wartością iloczynu skalarnego jest -1.00657107000000e-11 i nie udało się jej uzyskać żadnym z powyższych sposobów.
Kolejność wykonywania działań również ma wpływ na błędy obliczeniowe.

6 Wartości funkcji f = g zapisanych w różny sposób

Podane mamy poniższe funkcje:

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

Udowonijmy, że są one równe:
$g(x) = \frac{x^2}{\sqrt{x^2+1}+1} = \frac{x^2*(\sqrt{x^2+1}-1)}{(\sqrt{x^2+1}+1)*(\sqrt{x^2+1}-1)} = \frac{x^2*(\sqrt{x^2+1}-1)}{|x^2+1|-1} = \frac{x^2*(\sqrt{x^2+1}-1)}{x^2} = \sqrt{x^2+1} - 1 = f(x)$
Tak prezezntują się wartości funkcji f i g dla x=8^{-1},8^{-1},...,8^{-20}:

x	f(x)	g(x)
8^-1	0.0077822185373186414	0.0077822185373187065 30
8^-2	0.00012206286282867573	0.0001220628628287590 130
8^-3	1.9073468138230965e-6	1.907346813826566e-630
8^-4	2.9802321943606103e-8	2.9802321943606116e-830
8^-5	4.656612873077393e-10	4.6566128719931904e-1030
8^-6	7.275957614183426e-12	7.275957614156956e-1230
8^-7	1.1368683772161603e-13	1.1368683772160957e-1330
8^-8	1.7763568394002505e-15	1.7763568394002489e-1530
8^-9	0.0	2.7755575615628914e-1730
8^-10	0.0	4.336808689942018e-1930
8^-11	0.0	6.776263578034403e-2130
8^-12	0.0	1.0587911840678754e-2230
8^-13	0.0	1.6543612251060553e-2430
8^-14	0.0	2.5849394142282115e-2630
8^-15	0.0	4.0389678347315804e-2830

8^-16	0.0	6.310887241768095e-3030
8^-17	0.0	9.860761315262648e-3230
8^-18	0.0	1.5407439555097887e-3330
8^-19	0.0	2.407412430484045e-3530
8^-20	0.0	3.76158192263132e-3730

W funkcji f **odejmujemy liczby o bardzo podobnej wartości** przez występuje utrata precyzji. Wyniki funkcji g są bardziej wiarygodne. **Można więc wywnioskować, że przekształcenia funkcji pomagające unikać tego typu działań mogą zapewnić nam dokładniejsze wyniki**.

7 Przybliżona wartość pochodnej

Do obliczenia przybliżenia wartości pochodnej użyjemy poniższej funkcji:

```
function deriative(f, h, x0)
    return (Float64(f(x0 + h)) - Float64(f(x0)))/ Float64(h)
end
```

Funkcja f oraz jej pochodna prezentują się następująco:

$$f(x) = sinx + cos3x$$
$$f'(x) = cosx - 3sinx$$

Za pomocą poniższej pętli sprawdzimy jaka jest przybliżona wartość pochodnej w punkcie $x_0 = 1$ i jaki jest błąd przybliżenia $|f_{tilde}(x_0) - f'(x_0)|$ dla $h = 2^{-0}, 2^{-1}, ..., 2^{-54}$:

```
for i in 0:54
    # przybliżona wartość pochodnej w punkcie x0 = 1
    f_tilde = deriative(f, Float64(2.0)^-i, x0)
    println("2^-$i\t\t$f_tilde\t\t", abs(f_tilde - real_val))
end
```

Otrzymane wyniki:

h	f_tilde	|f_tilde - f'|
2^-0	2.0179892252685967	1.9010469435800585
2^-1	1.8704413979316472	1.753499116243109
2^-2	1.1077870952342974	0.9908448135457593
2^-3	0.6232412792975817	0.5062989976090435
2^-4	0.3704000662035192	0.253457784514981
2^-5	0.24344307439754687	0.1265007927090087
2^-6	0.18009756330732785	0.0631552816187897
2^-7	0.1484913953710958	0.03154911368255764
2^-8	0.1327091142805159	0.015766832591977753
2^-9	0.1248236929407085	0.007881411252170345

2^-10	0.12088247681106168	0.0039401951225235265
2^-11	0.11891225046883847	0.001969968780300313
2^-12	0.11792723373901026	0.0009849520504721099
2^-13	0.11743474961076572	0.0004924679222275685
2^-14	0.11718851362093119	0.0002462319323930373
2^-15	0.11706539714577957	0.00012311545724141837
2^-16	0.11700383928837255	6.155759983439424e-5
2^-17	0.11697306045971345	3.077877117529937e-5
2^-18	0.11695767106721178	1.5389378673624776e-5
2^-19	0.11694997636368498	7.694675146829866e-6
2^-20	0.11694612901192158	3.8473233834324105e-6
2^-21	0.1169442052487284	1.9235601902423127e-6
2^-22	0.11694324295967817	9.612711400208696e-7
2^-23	0.11694276239722967	4.807086915192826e-7
2^-24	0.11694252118468285	2.394961446938737e-7
2^-25	0.116942398250103	1.1656156484463054e-7
2^-26	0.11694233864545822	5.6956920069239914e-8
2^-27	0.11694231629371643	3.460517827846843e-8
2^-28	0.11694228649139404	4.802855890773117e-9
2^-29	0.11694222688674927	5.480178888461751e-8
2^-30	0.11694216728210449	1.1440643366000813e-7
2^-31	0.11694216728210449	1.1440643366000813e-7
2^-32	0.11694192886352539	3.5282501276157063e-7
2^-33	0.11694145202636719	8.296621709646956e-7
2^-34	0.11694145202636719	8.296621709646956e-7
2^-35	0.11693954467773438	2.7370108037771956e-6
2^-36	0.116943359375	1.0776864618478044e-6
2^-37	0.1169281005859375	1.4181102600652196e-5
2^-38	0.116943359375	1.0776864618478044e-6
2^-39	0.11688232421875	5.9957469788152196e-5
2^-40	0.1168212890625	0.0001209926260381522
2^-41	0.116943359375	1.0776864618478044e-6
2^-42	0.11669921875	0.0002430629385381522
2^-43	0.1162109375	0.0007313441885381522
2^-44	0.1171875	0.0002452183114618478
2^-45	0.11328125	0.003661031688538152
2^-46	0.109375	0.007567281688538152
2^-47	0.109375	0.007567281688538152
2^-48	0.09375	0.023192281688538152
2^-49	0.125	0.008057718311461848
2^-50	0.0	0.11694228168853815
2^-51	0.0	0.11694228168853815
2^-52	-0.5	0.6169422816885382
2^-53	0.0	0.11694228168853815
2^-54	0.0	0.11694228168853815

Najmniejszy błąd przybliżenia otrzymaliśmy dla $h = 2^{-28}$. Wraz z dalszym zmniejszaniem wartości h tracimy na dokładności w odejmowaniu co zwiększa błąd przebliżenia.

Jak zachowują się wartości 1+h?

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

2^-39	1.000000000001819	-0.1485215117923363
2^-40	1.0000000000009095	-0.14852151179244266
2^-41	1.0000000000004547	-0.14852151179249573
2^-42	1.0000000000002274	-0.14852151179252238
2^-43	1.0000000000001137	-0.1485215117925357
2^-44	1.0000000000000568	-0.14852151179254225
2^-45	1.0000000000000284	-0.1485215117925457
2^-46	1.0000000000000142	-0.14852151179254736
2^-47	1.000000000000007	-0.14852151179254813
2^-48	1.0000000000000036	-0.14852151179254858
2^-49	1.0000000000000018	-0.1485215117925487
2^-50	1.0000000000000009	-0.1485215117925489
2^-51	1.0000000000000004	-0.1485215117925489
2^-52	1.0000000000000002	-0.14852151179254902
2^-53	1.0	-0.1485215117925489
2^-54	1.0	-0.1485215117925489

Jak widać dla $h <= 2^{-53}$, $h + 1 = 1$, więc $f(h + 1) - f(1) = 0$. Dla coraz mniejszych wartości przebliżenie będzie równe 0. Dobieranie coraz mniejszych wartośći h powinno dawać nam coraz lepsze przybliżenie jednak przez ograniczenia precyzji nie dzieje się tak i utrata bitów w odejmowaniu zwiększa błąd przybliżenia.