Obliczenia Naukowe,
Laboratorium, Lista 1

Jerzy Wroczyński

2020-10-25

1 Zadanie 1.

1.1 Pod-zadanie 1.1.

1.1.1 Opis problemu

Epsilonem maszynowym (*macheps*) nazywamy najmniejszą liczbę w arytmetyce *fl* większą od 0 będącą odległością między 1 a następną liczbą. Naszym zadaniem jest jej wyznaczenie.

1.1.2 Rozwiązanie

Zadanie dzieli się na kilka mniejszych pod-zadań wyznaczania tej wartości osobno dla arytmetyk `Float16`, `Float32` oraz `Float64`. W języku `Julia` mamy dwa sposoby na uzyskanie takiej liczby:

- metoda iteracyjna (kod źródłowy w pliku `ex-1.1.jl`)
- wykonanie wbudowanej funkcji `eps`

W przypadku języka `C` musimy spojrzeć na predefiniowane globalne zmienne w pliku nagłówkowym `float.h`.

Zaobserwowane wartości:

- `Float16`:
  - metoda iteracyjna $= 9.77 \cdot 10^{-4}$
  - `eps(Float16)` $= 9.77 \cdot 10^{-4}$
- `Float32`:
  - metoda iteracyjna $= 1.1920929 \cdot 10^{-7}$
  - `eps(Float32)` $= 1.1920929 \cdot 10^{-7}$
  - `float.h: __FLT32_EPSILON__` $= 1.1920928955078125 \cdot 10^{-7}$
- `Float64`:
  - metoda iteracyjna $= 2.220446049250313 \cdot 10^{-16}$
  - `eps(Float64)` $= 2.220446049250313 \cdot 10^{-16}$
  - `float.h: __FLT64_EPSILON__` $= 2.22044604925031308084726333618164062 \cdot 10^{-16}$

Jak widać, we wszystkich przypadkach uzyskania żądanych stałych mamy te same wartości. Co oznacza, że nasza metoda iteracyjna jest poprawna.

1.2 Pod-zadanie 1.2.

1.2.1 Opis problemu

Liczbę maszynową η nazywamy najmniejszą liczbę w arytmetyce *fl* niebędącą zerem. Naszym zadaniem jest jej wyznaczenie.

1.2.2 Rozwiązanie

Tak jak w poprzednim pod-zadaniu, patrzymy na trzy różne arytmetyki `Float16`, `Float32` oraz `Float64`. Żeby uzyskać nasz wynik, wystarczy nieco zmodyfikować program użyty w poprzednim pod-zadaniu.

- `Float16`
  - metoda iteracyjna $= 6 \cdot 10^{-8}$
  - `nextfloat(fl(0.0))` $= 6 \cdot 10^{-8}$
- `Float32`
  - metoda iteracyjna $= 1 \cdot 10^{-45}$
  - `nextfloat(fl(0.0))` $= 1 \cdot 10^{-45}$
- `Float64`
  - metoda iteracyjna $= 5 \cdot 10^{-324}$
  - `nextfloat(fl(0.0))` $= 5 \cdot 10^{-324}$

Jak widać, wartości wyznaczone metodą iteracyjną nie różnią się od tych, wbudowanych w język `Julia` co oznacza, że metoda ta jest prawidłowa.

1.3 Pod-zadanie 1.3.

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

Obie te liczby mają taką samą wartość, jednakże nie opisują dokładnie tego samego. Liczba ϵ opisuje największy możliwy błąd względny w przypadku zaokrąglania liczby rzeczywistej. W zasadzie liczba `macheps` opisuje również taki największy możliwy błąd względny, jaki może się pojawić wokół liczby 1.0.

1.4 Pod-zadanie 1.4.

Jak związek ma liczba η z liczbą MIN_{sub} z wykładu?

Obie te liczby opisują tę samą własność — najmniejsza liczba w danej arytmetyce *fl*, która jest większa od 0 (bez normalizacji tej liczby).

1.5 Pod-zadanie 1.5.

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

Wynik programu `ex-1.5.jl`:

- `floatmin(Float32)` $= 1.1754944 \cdot 10^{-38}$
- `floatmin(Float64)` $= 2.2250738585072014 \cdot 10^{-308}$

Odpowiadające sobie wartości mają taki sam rząd wielkości i są prawie równe sobie. Opisują tę samą rzecz — najmniejsza liczba rzeczywista większa od zera, przy czym żądana wartość jest znormalizowana.

1.6 Pod-zadanie 1.6.

1.6.1 Opis problemu

Należy uzyskać wartość liczby granicznej MAX dla wszystkich typów zmiennopozycyjnych
`Float16`, `Float32` oraz `Float64`.

1.6.2 Rozwiązanie

Zadanie dzieli się na kilka mniejszych pod-zadań wyznaczania tej wartości osobno dla arytmetyk `Float16`, `Float32` oraz `Float64`. W języku `Julia` mamy dwa sposoby na uzyskanie takiej liczby:

- metoda iteracyjna (kod źródłowy w pliku `ex-1.6.jl`)
- wykonanie wbudowanej funkcji `floatmax`

Zaobserwowane wartości:

- `Float16`:
  - metoda iteracyjna $= 6.5 \cdot 10^4$
  - `floatmax(Float16)` $= 6.55 \cdot 10^4$
- `Float32`:
  - metoda iteracyjna $= 3.4028235 \cdot 10^{38}$
  - `eps(Float32)` $= 3.4028235 \cdot 10^{38}$
  - wartość podana na wykładzie: $3.4 \cdot 10^{38}$
- `Float64`:
  - metoda iteracyjna $= 1.7976931348623157 \cdot 10^{308}$
  - `eps(Float64)` $= 1.7976931348623157 \cdot 10^{308}$
  - wartość podana na wykładzie: $1.8 \cdot 10^{308}$

Jak widać, wszystkie wartości się zgadzają, co oznacza, że zastosowana metoda iteracyjna jest poprawna.

1.6.3 Opis metody iteracyjnej

(kod źródłowy w pliku `ex-1.6.jl`)

1. Wypełniamy mantysę przy pomocy funkcji `sum_`, która po kolei dodaje kolejne potęgi dwójki.
2. Następnie mnożymy otrzymaną liczbę przez 2 w pętli aż nie otrzymamy nieskończoności w rozumieniu danej arytmetyki (wykorzystujemy funkcję `isinf` do sprawdzania liczb)
3. Bierzemy jako wynik liczbę poprzedzającą tę, która jest już nieskończonością w tym ciągu generowanym przez tę pętlę.

2 Zadanie 2.

Wykonując polecenie `./ex-2.jl` otrzymujemy wyniki wyrażenia w arytmetykach `Float16`, `Float32` oraz `Float64`.

- `Float16`:
  - metoda Kahana $= 9.77 \cdot 10^{-4}$
  - `eps(Float16)` $= 9.77 \cdot 10^{-4}$
- `Float32`:
  - metoda Kahana $= 1.1920929 \cdot 10^{-7}$
  - `eps(Float32)` $= 1.1920929 \cdot 10^{-7}$
- `Float64`:
  - metoda Kahana $= 2.220446049250313 \cdot 10^{-16}$
  - `eps(Float64)` $= 2.220446049250313 \cdot 10^{-16}$

Metoda Kahana działa — wartości zwracane przez funkcję `kahans_eps` są równe wartościom ϵ odpowiednich arytmetyk.

3 Zadanie 3.

3.1 Pod-zadanie 3.1.

3.1.1 Opis problemu

Należy sprawdzić eksperymentalnie (w języku *Julia*), że w arytmetyce `Float64` każdą liczbę $x \in [1; 2]$ możemy zapisać w postaci

$$x = 1 + k \cdot \delta \tag{1}$$

gdzie $\delta = 2^{-52},\ k \in \{0, 1, 2, \ldots, 2^{52}\}$.

3.1.2 Rozwiązanie

(kod źródłowy w pliku `ex-3.1.jl`)

Program dołączony do tego zadania pokazuje przy pomocy ciągów jedynek i zer reprezentacje poszczególnych liczb w arytmetyce `Float64` w przedziale $[1; 2]$. Zmienna `starting_point` zmienia, od którego miejsca zacząć generować liczby z przedziału $[1; 2]$.

- `0011111111110000000000000000000000000000000000000000000000000000`
- `0011111111110000000000000000000000000000000000000000000000000001`
- `0011111111110000000000000000000000000000000000000000000000000010`
- `0011111111110000000000000000000000000000000000000000000000000011`
- `0011111111110000000000000000000000000000000000000000000000000100`
- `0011111111110000000000000000000000000000000000000000000000000101`

- 0011111111110000000000000000000000000000000000000000000000000110
- 0011111111110000000000000000000000000000000000000000000000000111
- 0011111111110000000000000000000000000000000000000000000000001000
- 0011111111110000000000000000000000000000000000000000000000001001
- 0011111111110000000000000000000000000000000000000000000000001010

Widać po ciągach bitowych (należy skupić się nad końcem ciągów), że zachodzi odliczanie wszystkich liczb z tego przedziału po kolei. Można to zjawisko porównać do zwykłego liczenia liczb naturalnych w systemie binarnym.
To samo zjawisko zachodzi dla innych części tego przedziału.

Zaprezentowane dowody empiryczne skłaniają do wniosku, że wyrażenie 1 jest prawdziwe.

3.2 Pod-zadanie 3.2.

(kod źródłowy w pliku `ex-3.2.jl`)

Tym razem rozważamy przedział $\left[\frac{1}{2}; 1\right]$. Wprowadzamy modyfikacje do programu z pod-zadania 3.1:

- zmieniamy początkową wartość `x` na `0.5`
- zmieniamy wartość δ na 2^{-53}

i zauważamy, że ponownie mamy to samo zjawisko — generowanie kolejnych liczb z zadanego przedziału.

- 0011111111100000000000000000000001000000000000000000000000001010
- 0011111111100000000000000000000001000000000000000000000000001011
- 0011111111100000000000000000000001000000000000000000000000001100
- 0011111111100000000000000000000001000000000000000000000000001101
- 0011111111100000000000000000000001000000000000000000000000001110
- 0011111111100000000000000000000001000000000000000000000000001111

Na powyższych wynikach jest pokazana po prostu inna część tego przedziału — zachodzi takie samo zjawisko.

Czyli liczba $x \in \left[\frac{1}{2}; 1\right]$ może zostać zapisana w następujący sposób

$$x = 0.5 + k \cdot \delta$$

gdzie $k \in \{0, 1, 2, \ldots, 2^{53}\}$, $\delta = 2^{-53}$.

3.2.1 Pod-zadanie 3.3.

(kod źródłowy w pliku `ex-3.3.jl`)

W przypadku przedziału $[2; 4]$ mamy dokładnie taką samą sytuację, przy czym zaczynamy oczywiście od 2, kiedy $\delta = 2^{-51}$.

4 Zadanie 4.

4.1 Opis problemu

($fl = \texttt{Float64}$)

1. Należy wyznaczyć liczbę $1 < x < 2$, taką, że $fl(x \cdot fl(1/x)) \neq 1$.

2. Należy znaleźć najmniejszą taką liczbę.

4.2 Rozwiązanie

Wynik programu `ex-4.jl` spełnia oba podpunkty zadania. Program zaczyna od liczby $1 + \delta$ (δ taka sama jak w zadaniu 3.1) i przegląda wszystkie liczby po kolei aż nie zostanie spełniony warunek `fl( x * fl(1/x)) != 1`.

Liczba znaleziona przez program wynosi 1.000001417355248.

5 Zadanie 5.

5.1 Opis problemu

Należy napisać program liczący iloczyn skalarny dwóch wektorów przy pomocy czterech metod w dwóch arytmetykach (`Float32` oraz `Float64`).

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

5.2 Rozwiązanie

(kod źródłowy w pliku `ex-5.jl`)

Aby uruchomić program, należy wykonać polecenie `./ex-5.jl` — wyświetlą się instrukcje mówiące, jakie argumenty są przyjmowane przez program.

Wyniki:

- `Float32`
  - metoda *„w przód”* $= -0.2499443$
  - metoda *„w tył”* $= -0.2043457$
  - od **największego** do *najmniejszego* $= -0.25$
  - od *najmniejszego* do **największego** $= -0.25$
- `Float64`
  - metoda *„w przód”* $= 1.0251881368296672e - 10$
  - metoda *„w tył”* $= -1.5643308870494366e - 10$
  - od **największego** do *najmniejszego* $= 0$
  - od *najmniejszego* do **największego** $= 0$

Faktyczna wartość wynosi $-1.00657107000000 \cdot 10^{-11}$.

Każda z metod, nieważne w jakiej arytmetyce, zwraca nieprawidłową wartość. Jedynie w arytmetyce `Float64` mamy nieco zbliżone do rzeczywistości wyniki — jednak nadal nie są one satysfakcjonujące.

6 Zadanie 6.

6.1 Opis problemu

Należy policzyć wartości funkcji

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

dla argumentów $x = 8^{-1}, 8^{-2}, 8^{-3}, \dots$ oraz porównać wyniki.

6.2 Rozwiązanie

Wyniki:

$$f(8^{-1}) = 0.0077822185373186414$$
$$g(8^{-1}) = 0.0077822185373187065$$
$$f(8^{-2}) = 0.00012206286282867573$$
$$g(8^{-2}) = 0.00012206286282875901$$
$$f(8^{-3}) = 1.9073468138230965 \cdot 10^{-6}$$
$$g(8^{-3}) = 1.907346813826566 \cdot 10^{-6}$$
$$f(8^{-4}) = 2.9802321943606103 \cdot 10^{-8}$$
$$g(8^{-4}) = 2.9802321943606116 \cdot 10^{-8}$$
$$f(8^{-5}) = 4.656612873077393 \cdot 10^{-10}$$
$$g(8^{-5}) = 4.6566128719931904 \cdot 10^{-10}$$
$$f(8^{-6}) = 7.275957614183426 \cdot 10^{-12}$$
$$g(8^{-6}) = 7.275957614156956 \cdot 10^{-12}$$
$$f(8^{-7}) = 1.1368683772161603 \cdot 10^{-13}$$
$$g(8^{-7}) = 1.1368683772160957 \cdot 10^{-13}$$
$$f(8^{-8}) = 1.7763568394002505 \cdot 10^{-15}$$
$$g(8^{-8}) = 1.7763568394002489 \cdot 10^{-15}$$
$$f(8^{-9}) = 0.0$$
$$g(8^{-9}) = 2.7755575615628914e{-}17$$
$$f(8^{-10}) = 0.0$$
$$g(8^{-10}) = 4.336808689942018e{-}19$$
$$f(8^{-11}) = 0.0$$
$$g(8^{-11}) = 6.776263578034403e{-}21$$
$$f(8^{-12}) = 0.0$$
$$g(8^{-12}) = 1.0587911840678754e{-}22$$

Uruchamiając program `ex-6.jl` widzimy, że faktycznie wartości się różnią.
Różnica oczywiście wynika z tego, że funkcje prezentują dwa różne sposoby policzenia tego samego. I właśnie to, że funkcja g wykonuje większą liczbę działań, jest tutaj kluczowe. Większa liczba działań oznacza większą liczbę popełnianych błędów w zaokrągleniach.

Jednakże możemy zauważyć jeszcze jedną rzecz. Otóż przy wartości 8^{-9} argumentu funkcja f zwraca już wartość 0, kiedy funkcja g nadal zwraca pewne wartości *relatywnie podobne* do tych dla większych argumentów.

Trudno jest powiedzieć, która funkcja jest bardziej *wiarygodna*, jednakże moim zdaniem funkcja f wygrywa prostszą strukturą działań, jest ich po prostu mniej, czyli mniej możliwości popełnionych błędów.

7 Zadanie 7.

7.1 Opis problemu

Należy obliczyć wartość pochodnej funkcji $f(x) = \sin x + \cos 3x$ przy pomocy wzoru na przybliżoną pochodną

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

oraz porównać z faktyczną wartością pochodnej.

- $h = 2^{-n}$, dla $n = 0, 1, 2, \dots, 54$
- błędy mierzymy przy pomocy: $|f'(x_0) - f'\tilde{(x_0)}|$

7.2 Rozwiązanie

(kod źródłowy w pliku `ex-7.jl`)

Faktyczną wartość pochodnej możemy obliczyć przy pomocy funkcji $f'(x) = \cos x - 3\sin 3x$.
Żeby zauważyć moment, w którym zmniejszanie wartości h nie poprawia przybliżenia wartości pochodnej, patrzymy na wartość błędów (odchylenia przybliżenia od faktycznej wartości pochodnej).
W programie zakres dla liczby n został ograniczony do przedziału $[-25; -32]$ (potęga 2 w liczbie h) w celu ułatwienia odczytania, w którym miejscu zachodzi to zjawisko.

- `n = -26`
```
h                          = 1.4901161193847656e-8
f(x + h) - f(x)            = 1.7425766385414931e-9
\tilde{f'}(x)              = 0.11694233864545822
f'(x)                      = 0.11694228168853815
|f'(x) - \tilde{f'}(x)|    = 5.6956920069239914e-8
```

- `n = -27`
```
h                          = 7.450580596923828e-9
f(x + h) - f(x)            = 8.712881527372929e-10
\tilde{f'}(x)              = 0.11694231629371643
f'(x)                      = 0.11694228168853815
|f'(x) - \tilde{f'}(x)|    = 3.460517827846843e-8
```

- `n = -28`
```
h                          = 3.725290298461914e-9
f(x + h) - f(x)            = 4.35643965346344e-10
\tilde{f'}(x)              = 0.11694228649139404
f'(x)                      = 0.11694228168853815
|f'(x) - \tilde{f'}(x)|    = 4.802855890773117e-9
```

- `n = -29`
```
h                          = 1.862645149230957e-9
f(x + h) - f(x)            = 2.1782187165086953e-10
\tilde{f'}(x)              = 0.11694222688674927
f'(x)                      = 0.11694228168853815
|f'(x) - \tilde{f'}(x)|    = 5.480178888461751e-8
```

- `n = -30`
```
h                          = 9.313225746154785e-10
f(x + h) - f(x)            = 1.0891088031428353e-10
\tilde{f'}(x)              = 0.11694216728210449
f'(x)                      = 0.11694228168853815
|f'(x) - \tilde{f'}(x)|    = 1.1440643366000813e-7
```

Najniższą wartość błędu otrzymujemy dla $n = -28$, dalsze wartości są tylko coraz większe.
Moment, w którym opisywane zjawisko zachodzi rząd wielkości liczb w liczniku i w mianowniku ułamka (będącego we wzorze aproksymacji pochodnej) jest bardzo niski (-10). Możliwe, że dochodzimy do granic możliwości dokładnego określenia, jaki powinien być wynik tego ułamka przez to, że te liczby są tak małe.

W przypadku kiedy przyjmujemy $x = 1 + h$ zamiast $x = 1$ moment, w którym zachodzi to zjawisko jest przesunięty do momentu $n = -30$.