Lista 1

Adrian Mucha 236526

20 października 2018

1 Zadanie 1

1.1 Machine epsilon

1.1.1 Problem

Używając języka `Julia`, napisać program wyznaczający iteracyjnie epsilony maszynowe. Epsilonem maszynowym nazywamy najmniejszą liczbę *macheps* > 0 taką, że *fl(1.0 + macheps) > 1.0* dla każdego typu zmiennopozycyjnego (`Float16, Float32, Float64`).

1.1.2 Rozwiązanie

Znalezienie epsilonu maszynowego opisałem w kilku krokach:

1. przypisz do wybranego typu zmiennej $epsilon = 1.0$
2. wykonuj dzielenie zmiennej *epsilon* przez 2 (przesunięcie bitowe w prawo), dopóki $1.0 + epsilon \neq 1.0$

Po tym jak pętla zostanie zakończona, w zmiennej *epsilon* znajdziemy najmniejszą wartość większą od zera - `macheps`.

1.1.3 Obserwacje

Im mniejsza wartość epsilona maszynowego, tym większa jest względna prezycja obliczeń.
Dla potwierdzenia, wykonajmy dodawanie $1.0 + macheps$ i przyjrzymy się zapisowi bitowemu tak otrzymanej liczby (dla `Float32`):

0 01111111 00000000000000000000001

Jest to więc liczba, którą możemy zapisać w postaci $2^0 \cdot 1.m$; m jest minimalną wartością (epsilonem), czyli wyznacza precyzję arytmetyki.

Typ	*macheps*	eps(typ)	float.h
Float16	0.000977	0.000977	—
Float32	$1.1920929e-7$	$1.1920929e-7$	$1.192093e-07$
Float64	$2.220446049250313e-16$	$2.220446049250313e-16$	$2.220446e-16$

Tabela 1: Wyniki wywołań poszczególnych funkcji dla wybranych typów wraz z danymi z pliku `float.h` języka C

1.1.4 Wyniki

Wyniki z Tabeli 1 zgadzają się z wartościami funkcji wbudowanych oraz tymi zawartymi w pliku nagłówkowym `float.h`.

1.2 Eta

1.2.1 Problem

Znaleźć iteracyjnie liczbę *eta* taką, że $eta > 0.0$ dla wszystkich typów zmiennopozycyjnych zgodnych ze standarded `IEEE 754`.

1.2.2 Rozwiązanie

Algorithm 1 Iteracyjne szukanie liczby *eta*

1: $x \leftarrow 1.0$
2: **while** $x/2 \neq 0.0$ **do**
3: $\quad x \leftarrow x/2$
4: **end while**
5: **return** x

Uzyskana w ten sposób liczba x zapisana bitowo w formacie `Float32` przyjmuje postać następującą:

$$\begin{gathered} 0\ 00000000\ 00000000000000000000001 \\ = 1.0e-45 \end{gathered}$$

Warto wspomnieć, że tego typu liczbę, której `cechą` są same zera - nazywamy nieznormalizowaną (subnormal).

1.2.3 Wyniki

Tabela 2 przedstawia wyniki dla następujących typów zmiennopozycyjnych:

Typ	*eta*	*nextfloat(0.0)*
Float16	$6.0e-8$	$6.0e-8$
Float32	$1.0e-45$	$1.0e-45$
Float64	$5.0e-324$	$5.0e-324$

Tabela 2: Wartości *eta* według typów zmiennoprzecinkowych

1.3 MAX

1.3.1 Problem

Napisać program, wyznaczający iteracyjnie liczbę MAX dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem `IEEE 754`

1.3.2 Rozwiązanie

W pętli `while` znajdujemy maksymalną cechę liczby, a następnie mnożymy przez największą możliwą mantysę z przedziału $[1, 2)$.

Algorithm 2 Iteracyjne szukanie MAX

```
x ← 1.0
while isinf(x · 2.0) == false do
    x ← x · 2.0
end while
x ← x · (2.0 − eps)
return x
```

1.3.3 Wyniki

Otrzymane wyniki w Tabeli 3 zgadzają się zarówno z funkcjami wbudowanymi jak i wartościami ze standardowego pliku nagłowkowego `float.h` języka C.

Typ	*MAX*	*realmax*	float.h
Float16	$6.55e-4$	$6.55e-4$	—
Float32	$3.4028235e38$	$3.4028235e38$	$3.402823e+38$
Float64	$1.7976931348623157e308$	$1.7976931348623157e308$	$1.797693e+308$

Tabela 3: Wartości *MAX* według typów zmiennoprzecinkowych

2 Zadanie 2 - Kahan

2.1 Problem

Sprawdzić eksperymentalnie w języku `Julia` słusznośc twierdzenia Kahana: Epsilon maszynowy można otrzymać obliczając wyrażenie $3(4/3-1)-1$ w arytmetyce zmiennopozycyjnej.

2.2 Rozwiązanie

`FloatXX` $\in$ {`Float16`, `Float32`, `Float64`}

Algorithm 3 Kahan Epsilon

```
1: a ← FloatXX(3.0)
2: b ← FloatXX(4.0)
3: c ← FloatXX(1.0)
4: x ← FloatXX(a((b/a) − c) − c)
5: return x
```

2.3 Wyniki

Typ	*KAHAN*	*eps*
Float16	-0.000977	0.000977
Float32	$1.1920929e-7$	$1.1920929e-7$
Float64	$-2.220446049250313e-16$	$2.220446049250313e-16$

Tabela 4: Wartości *Twierdzenia Kahana* według typów zmiennoprzecinkowych

2.4 Obserwacja

Możemy zauważyć, że `Twierdzenie Kahana` zdaje się działać prawidłowo jedynie dla typu zmiennopozycyjnego `Float32`. W innych przypadkach wartość posiada znak odwrotny, lecz gdyby zastosować wartość bezwzględną wyników, to byłyby identyczne.

3 Zadanie 3 - Rozkład liczb zmiennopozycyjnych

3.1 Problem

Sprawdź eksperymentalnie w języku Julia, że w arytmetyce `Float64` (arytmetyce double w standarcie `IEEE 754`) liczby zmiennopozycyjne są równomiernie rozmieszczone w $[1, 2]$ z krokiem $\delta = 2^{-52}$. Innymi słowy, każda liczba zmiennopozycyjna x pomiędzy 1 i 2 może być przedstawione następująco $x = 1 + k\delta$ w tej arytmetyce, gdzie $k = 1, 2, ..., 2^{52} - 1$ i $\delta = 2^{-52}$.

3.2 Rozwiązanie

Opis algorytmu eksperymentalnego, gdzie:
$k \in \{1, 2, ..., 2^{52} - 1\}$,
$\delta \leftarrow 2^{-52}$

1. utworzenie zmiennej $x \leftarrow 1.0 + k \cdot \delta$
2. wypisz bitową reprezentację x
3. $k \leftarrow k + 1$

3.3 Obserwacje

3.3.1 Przedział [1,2]

Liczba	Zapis bitowy
1.0000000000000002	0011111111110000000000000000...0000000000000001
1.0000000000000004	0011111111110000000000000000...0000000000000010
1.0000000000000007	0011111111110000000000000000...0000000000000011
⋮	⋮
1.9999999999999993	0011111111111111111111111111...1111111111111101
1.9999999999999996	0011111111111111111111111111...1111111111111110
1.9999999999999998	0011111111111111111111111111...1111111111111111

Tabela 5: Liczby i ich bitowe odpowiedniki w zakresie [1,2]

Patrząc na zapis binarny liczby, możemy zauważyć, że przy każdej iteracji, *mantysa* x jest modyfikowana tak, jakbyśmy dodali do niej binarną jedynkę. To znaczy, że te liczby są równomiernie rozmieszczone w $[1, 2]$ z krokiem $\delta = 2^{-52}$.

3.3.2 Przedział [0.5, 1]

Liczba	Zapis bitowy
0.5000000000000001	001111111110000000000000000000...0000000000000010
0.5000000000000002	001111111110000000000000000000...0000000000000100
0.5000000000000003	001111111110000000000000000000...0000000000000110
⋮	⋮
0.9999999999999997	001111111110111111111111111111...1111111111111010
0.9999999999999998	001111111110111111111111111111...1111111111111100
0.9999999999999999	001111111110111111111111111111...1111111111111110

Tabela 6: Liczby i ich bitowe odpowiedniki w zakresie [0.5,1]

Wniosek jest taki, że liczby są tutaj rozmieszczone "gęściej", to znaczy, że $\delta = \frac{1}{2} \cdot 2^{-52} = 2^{-53}$. Obserwacja pochodzi z faktu, że dodanie do liczb z przedziału [0.5, 1], kolejnych wartości $k\delta$, gdzie ($\delta = 2^{-52}$) powoduje zmianę *mantysy* o "binarną dwójkę".

3.3.3 Przedział [2, 4]

Symetrycznie do przedziału [0.5, 1], otrzymamy rzadsze rozmieszczenie liczb, czyli $\delta = 2 \cdot 2^{-52} = 2^{-51}$.

3.3.4 Wniosek

Liczby zmiennopozycyjne są rozmieszczone nierównomiernie. Liczby bliskie zeru są rozmieszczone bardzo gęsto, natomiast im zakres jest większy, tym rzadziej one występują. Jest to następstwem tego, że `cecha` rośnie coraz szybciej, lecz ilość bitów `mantysy` jest taka sama, co skutkuje stałą pojemnością liczb.

4 Zadanie 4

4.1 Problem

Znajdź eksperymentalnie w arytmetyce `Float64` zgodnej ze standardem `IEEE 754 (double)` liczbę zmiennopozycyjną x w przedziale $1 < x < 2$, taką, że $x \cdot \frac{1}{x} \neq 1$; tj. $fl(xfl(1/x)) \neq 1$ (napisz program w języku Julia znajdujący tę liczbę).

4.2 Rozwiązanie

Do znalezienia rozwiązania posłużymy się funkcjami języka `Julia`, a mianowicie `nextfloat(`z`)`, która zwraca kolejną liczbę zmiennopozycyjną, większą od z.

$x \leftarrow nextfloat(1.0)$
while $x \cdot \frac{1}{x} \neq 1.0 \wedge x < 2.0$ **do**
    $x \leftarrow nextfloat(x)$
end while
return x

4.3 Wynik

Wynik to: 1.000000057228997, co jednocześnie jest najmniejszą taką liczbą w tym przedziale.

4.4 Wnioski

Takie błędy, wynikające z ograniczonej precyzji standardu `IEEE 754` mogą prowadzić do nagromadzenia się błędu obliczeń i dać horrendalnie niepoprawne wyniki.

5 Zadanie 5 - Sumy

5.1 Problem

Napisz program w języku Julia realizujący następujący eksperyment obliczania iloczynu skalarnego 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 Wyniki

Podpunkt	`Float32`	`Float64`
Do przodu	-0.4999443	$1.0251881368296672e-10$
Od tyłu	-0.4543457	$-1.5643308870494366e-10$
Największe-Najmniejsze	-0.34720382	0.0
Najmniejsze-Największe	-0.34720382	0.0

Tabela 7: Wyniki poszczególnych strategii sumowania

Podany w treści zadania dokładny wynik $-1.00657107000000010e^{-11}$ różni się od otrzymanych. Wszystkie wyniki są bliskie zeru, co wiąże się z faktem, że wektory ortogonalne generują duże błędy obliczeń.

6 Zadanie 6

6.1 Problem

Policz w języku `Julia` w arytmetyce `Float64` wartości następujących funkcji

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

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$

6.2 Rozwiązanie

Iteracyjne wywołanie funkcji $f(x)$ oraz $g(x)$ dla kolejnych argumentów.

6.3 Wyniki

6.4 Wnioski

Wyniki są zbliżone dla pierwszych 8 iteracji, lecz później, w pierwszej funkcji $f(x)$ błąd obliczeń jest ogromny. Jest to spowodowane redukcją cyfr znaczących przez odejmowanie od siebie bliskich wartości. W tych funkcjach mamy $\sqrt{x^2+1} \approx 1$ dla małych x. Dlatego w funkcja $f(x)$ może dawać fałszywe wyniki.

Parametr	$f(x)$	$g(x)$
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	$4.336808689942018e-19$

Tabela 8: Wyniki kolejnych iteracji funkcji $f(x)$ oraz $g(x)$

7 Zadanie 7

7.1 Problem

Przybliżoną wartość pochodnej $f(x)$ w punkcie x można obliczyć za pomocą następującego wzoru

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

Skorzystać z tego wzoru do obliczenia w języku Julia w arytmetyce Float64 przybliżonej wartości pochodnej funkcji $f(x) = sinx + cos3x$ w punkcie $x_0 = 1$ oraz błędów $|f'(x_0) - f(x_0)|$ dla $h = 2^{-n}(n = 0, 1, 2, ..., 54)$.

7.2 Rozwiązanie

Obliczenie (ręczne) pochodnej funkcji $f(x)$ $(g(x) = f'(x) = cos(x) - 3sin(3x))$. W kolejnych iteracjach pętli obliczam wartość pochodnej wg. podanego wzoru wyżej i błąd względem wyniku funkcji $g(x)$.

7.3 Wyniki

7.4 Wnioski

Od pewnego momentu zmniejszanie h nie pomaga ze względu na błędy dokładności operacji $1.0 + h$. Ten błąd jest nie tylko miejscowy, ale przekłada się na dalsze obliczenia tej funkcji, które mogą spowodować horrendalnie duży błąd względny. Najmniejszy błąd obserwujemy dla parametru $h = 2^{-28}$.

Parametr h	$f(x)$	Błąd
2^0	2.0179892252685967	1.9010469435800585
2^{-1}	1.8704413979316472	1.753499116243109
2^{-2}	1.1077870952342974	0.9908448135457593
$\vdots$	$\vdots$	$\vdots$
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	$1.1440643366000813e-7$
$\vdots$	$\vdots$	$\vdots$
2^{-52}	-0.5	0.6169422816885382
2^{-53}	0.0	0.11694228168853815
2^{-54}	0.0	0.11694228168853815

Tabela 9: Wyniki kolejnych iteracji funkcji $f(x)$ oraz $g(x)$