‘

Obliczenia naukowe
Lista 1

Dominik Kaczmarek, nr albumu 261757

24 października 2022

Spis treści

1 Rozpoznanie arytmetyki

1.1 Opis problemu

Napisanie programu, który przy użyciu iteracji obliczy:

a) *macheps* (*epsilon maszynowy*), czyli najmniejszą liczbę taką, że $fl(1.0 + macheps) > 1.0$

b) *eta* (*liczba maszynowa*), czyli najmniejszą liczbę taką, że $fl(eta) > 0.0$

c) MAX, czyli największą liczbę dla danej precyzji

dla wszystkich dostępnych typów zmiennopozycyjnych **Float16**, **Float32**, **Float64** zgodnych ze standardem IEEE 754. Otrzymane wyniki należy później porównać:

a) *macheps* z wartościami zwracanymi przez funkcje **eps**($Float16$), **eps**($Float32$), **eps**($Float64$), oraz z danymi zawartymi w pliku nagłówkowym `float.h` dowolnej instalacji języka C.

b) *eta* z wartościami zwracanymi przez funkcje: **nextfloat**($Float16(0.0)$), **nextfloat**($Float32(0.0)$), **nextfloat**($Float64(0.0)$), oraz z MIN_{sub}

c) MAX z wartościami zwracanymi przez funkcje: **floatmax**($Float16$), **floatmax**($Float32$), **floatmax**($Float64$), oraz z danymi zawartymi w pliku nagłówkowym `float.h` dowolnej instalacji języka C lub z danymi z wykładu.

1.2 Rozwiązanie

W każdym algorytmie zaczynamy od przypisaniu zmiennej x wartości 1.0 ($Float16(1)$, $Float32(1)$, $Float64(1)$). Następnie w pętli wykonujemy na zmiennej x następujące operacje:

a) *macheps* — dzielimy $x/2$ dopóki spełniony jest warunek $1 + x/2 > 1.0$. Po wyjściu z pętli $x = macheps$.

b) *eta* — dzielimy $x/2$ dopóki $x/2 > 0.0$. Po wyjściu z pętli $x = eta$.

c) MAX — mnożymy $2 \cdot x$ dopóki $2 \cdot x < \infty$. Po wyjściu z pętli wiemy że $2 \cdot x = \infty$, ale wciąż $x \neq MAX$. Tworzymy nową zmienną pomocniczą $y = x$. Teraz dzielimy $y/2$ i pod x przypisujemy $x = x + y$ dopóki $x < \infty$. Dopiero po zakończeniu tej pętli otzrymamy $x = MAX$.

Programy wyznaczające szukane wartości znajdują się w pliku *z1.jl*.

1.3 Wyniki

typ	macheps	eps(typ)	float.h
Float16	0.000977	0.000977	—
Float32	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$	$1.192093 \cdot 10^{-7}$
Float64	$2.220446049250313 \cdot 10^{-16}$	$2.220446049250313 \cdot 10^{-16}$	$2.220446 \cdot 10^{-16}$

Tabela 1: Zestawienie obliczonych *macheps* z wynikami zwracanymi przez funkcję *eps* oraz danymi z biblioteki `float.h`

typ	eta	nextfloat(typ(0.0))	MIN_{sub}
Float16	$6.0 \cdot 10^{-8}$	$6.0 \cdot 10^{-8}$	—
Float32	$1.0 \cdot 10^{-45}$	$1.0 \cdot 10^{-45}$	$1.4 \cdot 10^{-45}$
Float64	$5.0 \cdot 10^{-324}$	$5.0 \cdot 10^{-324}$	$4.9 \cdot 10^{-324}$

Tabela 2: Zestawienie obliczonego *eta* z wbudowaną funckją *nextfloat(typ(0.0))* oraz wartościami MIN_{sub} (podane na wykładzie).

typ	MAX	floatmax(typ)	float.h
Float16	$6.55 \cdot 10^4$	$6.55 \cdot 10^4$	—
Float32	$3.4028235 \cdot 10^{38}$	$3.4028235 \cdot 10^{38}$	$3.402823 \cdot 10^{38}$
Float64	$1.7976931348623157 \cdot 10^{308}$	$1.7976931348623157 \cdot 10^{308}$	$1.797693 \cdot 10^{308}$

Tabela 3: Zestawienie obliczonego MAX z wynikami zwracanymi przez funkcję $floatmax(typ)$ oraz danymi z biblioteki `float.h`.

typ	floatmin(typ)	MIN_{nor}
Float32	$1.1754944 \cdot 10^{-38}$	$1.2 \cdot 10^{-38}$
Float64	$2.2250738585072014 \cdot 10^{-308}$	$2.2 \cdot 10^{-308}$

Tabela 4: Zestawienie obliczonego MAX z wynikami zwracanymi przez funkcję $floatmax(typ)$ oraz danymi z biblioteki `float.h`.

1.4 Wnioski

- Wyniki otrzymane z iteracyjnych algorytmów pokrywają się z wartościami zwracanymi przez funkcje wbudowane w język *Julia*.
- Komputer nie jest w stanie przedstawić wszytskich liczb rzeczywistymi
- Im większa odległość od zera maszynowego tym mniej liczb znajduje się między pobliski liczbami całkowitymi
- Porównując *eta* z MIN_{sub} widzimy, że zgadza się tylko rząd wielkości
- *floatmin(typ)* zwraca najmniejszą bezwzględną liczbę dla zadanego typu
- Porównując *floatmin(typ)* z MIN_{nor} widzimy, że zgadza się tylko rząd wielkość
- *eta* i *nextfloat(typ(0.0))* mają postać zdenormalizowaną, natomiast *floatmin(typ)* zwraca wartość znormalizowaną
- $macheps = 2 \cdot \epsilon$

2 Epsilon maszynowy Kahana

2.1 Opis problemu

Obliczenie wyrażenia $3 \cdot (4/3 - 1) - 1$ dla wszystkich dostępnych typów zmiennopozycyjnych **Float16**, **Float32**, **Float64** i sprawdzenie czy pokrywa się z epsilonem maszynowym odpowiednich arytmetyk.

2.2 Rozwiązanie

Używamy funkcji *one*(*typ*) do uzykania liczby 1 każdego z zadanych typów zmiennopozycyjnych. Następnie obliczamy wyrażenie $3 \cdot (4/3 - 1) - 1$ dla każdego typu.

Programy wyznaczające szukane wartości znajdują się w pliku *z2.jl*.

2.3 Wyniki i Interpretacja

typ	macheps Kahana	eps(typ)
Float16	-0.0009765625	0.0009765625
Float32	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$
Float64	$-2.220446049250313 \cdot 10^{-16}$	$2.220446049250313 \cdot 10^{-16}$

Tabela 5: Zestawienie obliczonego *macheps* Kahana dla różnych typów wraz z poprawnymi wartościami.

Wartości bezwzględne wyników uzyskanych za pomocą obliczeń zgadzają się z poprawnymi epsilonami maszynowymi. Ujemne wyniki spowodowane są parzystością mantysy typów i faktem, że w tym wypadku w rozwinięciu dwójkowym liczby 4/3 na ostatniej pozycji mantysy znajduje się 0, a więc zgodnie z zasadą *"round to even"* liczba zaokrąglana jest z niedomiarem, co przy dalszych obliczeniach daje wynik ujemny.

2.4 Wnioski

Przez skończoną dokładność reprezentacji, niektóre równania dające w normalnej arytmetyce 0, w naszym przypadku zwracają wyniki różne od 0.

3 Rozmieszczenie liczb zmiennopozycyjnych

3.1 Opis problemu

Sprawdzić eksperymentalnie w języku Julia, że w arytmetyce Float64 (arytmetyce double w standarcie IEEE 754) liczby zmiennopozycyjne są równomiernie rozmieszczone w $[1, 2)$ z krokiem $\delta = 2^{-52}$. Sprawdzić jak rozmieszczone są liczby zmiennopozycyjne w przedziale $[\frac{1}{2}, 1)$, jak w przedziale $[2, 4)$ i jak mogą być przedstawione dla rozpatrywanego przedziału.

3.2 Rozwiązanie

a) Przedział $[1, 2)$
Mając $\delta = 2^{-52}$ dla każdego $k = 1, 2, ..., 2^{52} - 1$ liczymy wartość $x = 1 + k\delta$ i przedstawiamy x w postaci bitowej używając funkcji `bitstring(`x`)`.

b) Przedział $[2, 4)$
Korzystając z intuicji powiększamy deltę dwukrotnie (bo przedział $[2, 4)$ znajduje się dalej od zera niż $[1, 2)$), czyli $\delta = 2^{-51}$. Robimy to samo co w punkcie *a)* tylko dla $\delta = 2^{-51}$.

c) Przedział $[\frac{1}{2}, 1)$
Analogicznie, korzystając z intuicji tym razem pomniejszamy deltę dwukrotnie (bo przedział $[\frac{1}{2}, 1)$ znajduje się bliżej zera niż $[1, 2)$), czyli $\delta = 2^{-53}$. Robimy to samo co w punkcie *a)* tylko dla $\delta = 2^{-53}$.

3.3 Wyniki i interpretacja

$[\frac{1}{2}, 1]$ $\quad\quad$ $\delta = 2^{-53}$
0011111111100000000000000000000000000000000000000000000000000001
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000011
0011111111100000000000000000000000000000000000000000000000000100
$\vdots$
0011111111101111111111111111111111111111111111111111111111111100
0011111111101111111111111111111111111111111111111111111111111101
0011111111101111111111111111111111111111111111111111111111111110
0011111111101111111111111111111111111111111111111111111111111111
$[1, 2]$ $\quad\quad$ $\delta = 2^{-52}$
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
$\vdots$
0011111111111111111111111111111111111111111111111111111111111100
0011111111111111111111111111111111111111111111111111111111111101
0011111111111111111111111111111111111111111111111111111111111110
0011111111111111111111111111111111111111111111111111111111111111
$[2, 4]$ $\quad\quad$ $\delta = 2^{-51}$
0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
$\vdots$

0100000000001111111111111111111111111111111111111111111111111100
0100000000001111111111111111111111111111111111111111111111111101
0100000000001111111111111111111111111111111111111111111111111110
0100000000001111111111111111111111111111111111111111111111111111

Tabela 6: Prezentacja bitowa wybranych liczb w danych przedziałach.

Dzięki funkcji `bitstring(`x`)` możemy zauważyć, że przy inkrementacji zmiennej k w każdej z rubryk inkrementuje się wartość mantysy. Fakt, że sąsaidujące wiersze różnią się dokładnie o 1 mówi nam, że intuicja z mnożeniem lub dzieleniem bazowej $\delta = 2^{-52}$ (*machepsu*) była poprawna.

3.4 Wnioski

W arytmetyce **Float64** w każdym z przedziałów postaci $[2^k, 2^{k+1})$ dla $k \in \mathbb{C}$, znajduje się dokładnie 2^{52} liczb. Im większa odległość od *zera maszynowego* tym mniej liczb znajduje się w przedziałach postaci $[k, k+1)$

4 Nieodwracalność dzielenia

4.1 Opis problemu

Znaleźć eksperymentalnie w arytmetyce `Float64` dwie liczby:

a) x takiego, że $x \in (1,2) \wedge x \cdot (1/x) \neq 1$

b) x takiego, że $x > 0 \wedge x \cdot (1/x) \neq 1$

4.2 Rozwiązanie

a) Zaczynając od $x = Float64(1)$ zwiększamy x korzystjąc z funkcji `nextfloat(x)` do momentu kiedy $x \cdot (1/x) \neq 1$

b) Zaczynając od $x = Float64(0)$ zwiększamy x korzystjąc z funkcji `nextfloat(x)` do momentu kiedy $x \cdot (1/x) \neq 1$

4.3 Wyniki

a) Najmniejsza znaleziona liczba to x = 1.000000057228997
$x \cdot \frac{1}{x} = 0.9999999999999999$

b) Najmniejsza znaleziona liczba to x = $5.0 \cdot 10^{-324}$
$x \cdot \frac{1}{x} = \infty$

4.4 Wnioski

Zadanie pokazuje, że działania arytmetyczne na liczbach zmiennopozycyjnych mogą generować błędy związane z zaokrąglaniem wyliczonych wartości. Nawet z pozoru proste rachunki mogą dać nieprawidłowe wyniki. Użytkownik powinien maksymalnie upraszczać swoje algorytmy w celu zminimalizowania szansy wystąpienia błędu.

5 Iloczyn skalarny

5.1 Opis problemu

Obliczenie iloczynu skalarnego danych wektorów x i y z wykorzystaniem czterech różnych algorytmów sumowania dla typów `Float32` i `Float64`.

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

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

Algorytmy do sprawdzenia:

a) "w przód": $\sum_{i=1}^{n} x_i y_i$

b) "w tył": $\sum_{i=n}^{1} x_i y_i$

c) dodanie dodatnich liczb w porządku od największej do najmniejszej oraz ujemnych w porządku od najmniejszej do największej, a następnie dodanie do siebie obliczonych sum częściowych

d) metoda przeciwna do sposobu *3*

5.2 Rozwiązanie

Programy zawierające powyższe algorytmy znajdują się w pliku `z5.jl`.

5.3 Wyniki

typ	a	b	c	d
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	$1.0251881368296672 \cdot 10^{-10}$	$-1.5643308870494366 \cdot 10^{-10}$	0.0	0.0

Tabela 7: Iloczyn skalarny danych wektorów, wyliczony każdym z podanych algorytmami

Rzeczywisty iloczyn skalarny x i y wynosi $-1.00657107000000 \cdot 10^{-11}$. Żaden z otrzymanych wyników nie pasuje do prawidłowej wartości. Najbliższy do oryginału wynik dało zastosowanie algorytmu `a)` z arytmetyką `Float64`.

5.4 Wnioski

Eksperyment pokazał, że kolejność wykonywanych obliczeń drastycznie wpływa na otrzymywany wynik. Przy sumowaniu dwóch liczb z których jedna jest znacząco większa od drugiej, ta większa pochłania mniejszą przez wynik staje się mniej dokładny.

6 Przybliżenie funkcji

6.1 Opis problemu

Obliczenie w arytmetyce `Float64` wartości dwóch funkcji

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

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

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

6.2 Rozwiązanie

Iteracja po kolejnych potęgach ósemki i liczenie wartości f i g dla każdej z nich. Programy znajdują się w pliku `z6.jl`.

6.3 Wyniki i Interpretacja

Z analizy matematycznej wiemy, że funkcje f i g są sobie równe i dla $x \in 8^{-1}, 8^{-2}, \ldots, 8^{-8}$ ich wartości są rzeczywiście zbliżone. Jednak dla $x < 8^{-8}$ funkcja f zaczęła zwracać wartość 0.0, podczas gdy funkcja g jeszcze dla $x = 8^{-178}$ pokazuje wartość różną od zera ($1.6 \cdot 10^{-322}$). Różnice stanowi fakt, że w funkcji f odejmujemy 1 od pierwiastka $\sqrt{x^2 + 1}$ przez co f operuje na wartościach bardzo bliskich 0, co z kolei powoduje utratę cyfr znaczących.. Funkcja g nie generuje takiego błędu przez co jest dużo bardziej dokładna niż funkcja f.

x	$f(x)$	$g(x)$
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	$1.9073468138230965 \cdot 10^{-6}$	$1.907346813826566 \cdot 10^{-6}$
8^{-4}	$2.9802321943606103 \cdot 10^{-8}$	$2.9802321943606116 \cdot 10^{-8}$
$\vdots$	$\vdots$	$\vdots$
8^{-8}	$1.7763568394002505 \cdot 10^{-15}$	$1.7763568394002489 \cdot 10^{-15}$
8^{-9}	0.0	$2.7755575615628914 \cdot 10^{-17}$
$\vdots$	$\vdots$	$\vdots$
8^{-176}	0.0	$6.4758 \cdot 10^{-319}$
8^{-177}	0.0	$1.012 \cdot 10^{-320}$
8^{-178}	0.0	$1.6 \cdot 10^{-322}$
8^{-179}	0.0	0.0

Tabela 8: Wartości funkcji f i g dla kolejnych argumentów.

6.4 Wnioski

Obliczenia należy wykonywać w ten sposób, aby liczba cyfr znaczących przy kolejnych działaniach różniła się jak najmnniej. Pozwoli to uniknąć szybkiej kumulacji utarty dokładności.

7 Przybliżenie pochodnej

7.1 Opis problemu

Mając funkcję

$$f(x) = \sin x + \cos 3x$$

obliczyć przybliżone wartości pochodnej f w punkcie $x_0 = 1$ ze wzoru

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

przy $h \to 0$ i porównać je wynikami matematycznie wyznaczonej pochodnej

$$f'(x) = \cos(x) - 3 \cdot \sin(3x)$$

Obliczyć błąd $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$ $(n = 0, 1, 2, \ldots, 54)$.

7.2 Rozwiązanie

Program znajduje się w pliku `z7.jl`

7.3 Wyniki

Wyniki obliczeń dla poszczególnych wartości h przedstawiono poniżej (Tabela 9). Początkowo zmniejszanie wartości h przynosi oczekiwane skutki i błędy w liczeniu przybliżonej pochodnej są mniejsze, najdokładniejszy wynik uzyskano dla $h = 2^{-28}$. Dalsze zmniejszanie h nie poprawiło jednak dokładności obliczeń, wręcz przeciwnie, błędy zaczęły z powrotem rosnąć.

7.4 Wnioski

Przy prowadzeniu obliczeń należy wystrzegać się liczb bardzo bliskich zeru.

h	$\tilde{f}'(1)$	$\lvert f'(1) - \tilde{f}'(1)\rvert$	$1 + h$
2^{0}	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	1.8704413979316472	1.753499116243109	1.5
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-16}	0.11700383928837255	$6.155759983439424 \cdot 10^{-5}$	1.0000152587890625
2^{-17}	0.11697306045971345	$3.077877117529937 \cdot 10^{-5}$	1.0000076293945312
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-27}	0.11694231629371643	$3.460517827846843 \cdot 10^{-8}$	1.0000000074505806
2^{-28}	0.11694228649139404	$4.802855890773117 \cdot 10^{-9}$	1.0000000037252903
2^{-29}	0.11694222688674927	$5.480178888461751 \cdot 10^{-8}$	1.0000000018626451
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-36}	0.116943359375	$1.0776864618478044 \cdot 10^{-6}$	1.000000000014552
2^{-37}	0.1169281005859375	$1.4181102600652196 \cdot 10^{-5}$	1.000000000007276
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.11694228168853815	1.0
2^{-54}	0.0	0.11694228168853815	1.0

Tabela 9: Wartości funkcji f i g dla kolejnych argumentów.