Obliczenia naukowe
Lista 1

Miriam Jańczak 229761

Prowadzący: dr hab. Paweł Zieliński, prof. PWr

22.10.2017

Przedstawione tutaj zadania mają na celu przybliżenie arytmetyki zmiennopozycyjnej (**IEEE 754**), lepsze jej zrozumienie, a także przedstawienie różnego typu właściwości i zagrożeń związanych z wykonywaniem w niej obliczeń.

1 Rozpoznanie arytmetyki

1.1 Epsilon maszynowy

Epsilonem maszynowym *macheps* (ang. machine epsilon) nazywamy najmniejszą liczbę $macheps > 0$ taką, że $1.0 \oplus macheps > 1.0$.

1.1.1 Opis problemu

Iteracyjne wyznaczenie epsilonów maszynowych dla wszystkich typów zmiennopozycyjnych i porównanie ich z wartościami zwracanymi przez funkcję *eps* z języka Julia oraz z danymi zawartymi w pliku nagłówkowym `float.h` języka C.

1.1.2 Rozwiązanie

W celu wyznaczenia epsilonu maszynowego zastosowano następujący algorytm:

$macheps \leftarrow 1$
while $1 + macheps/2 > 1$ **do**
    $macheps \leftarrow macheps/2$
end while

1.1.3 Wyniki

Dla poszczególnych typów zmiennopozycyjnych otrzymano następujące wyniki (Tabela 1).

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

Tabela 1: Zestawienie obliczonego epsilonu maszynowego dla różnych typów wraz z poprawnymi wartościami.

1.1.4 Wnioski

Metoda iteracyjna obliczania epsilonu maszynowego jest poprawna, ponieważ wyniki uzyskane za jej pomocą pokrywają się z prawidłowymi wartościami. Otrzymane wyniki pokazują, że im większa jest precyzja arytmetyki, tym mniejszy jest epsilon maszynowy. Epsilon maszynowy jest ściśle związany z precyzją arytmetki zadaną wzorem 2^{-t-1}, gdzie t jest długością mantysy, jego wartość jest dwa razy większa (2^{-t}).

1.2 ETA

1.2.1 Opis problemu

Iteracyjne wyznaczenie liczby *eta* takiej, że $eta > 0.0$ dla wszystkich typów zmiennopozycyjnych i porównanie jej z wartościami zwracanymi przez funkcję *nextfloat* z języka Julia, a także liczbą MIN_{sub}.

1.2.2 Rozwiązanie

W celu wyznaczenia liczby *eta* zastosowano następujący algorytm:

$eta \leftarrow 1$
while $eta/2 > 0.0$ **do**
    $eta \leftarrow eta/2$
end while

1.2.3 Wyniki

Dla poszczególnych typów zmiennopozycyjnych otrzymano następujące wyniki (Tabela 2).

	eta	nextfloat(typ)	MIN_{sub}
Float16	$5.960464 \cdot 10^{-8}$	$5.960464 \cdot 10^{-8}$	—
Float32	$1.4012985 \cdot 10^{-45}$	$1.4012985 \cdot 10^{-45}$	$1.4012985 \cdot 10^{-45}$
Float64	$4.9406564584124654 \cdot 10^{-324}$	$4.9406564584124654 \cdot 10^{-324}$	$4.9406564584124654 \cdot 10^{-324}$

Tabela 2: Zestawienie obliczonego *eta* dla różnych typów wraz z poprawnymi wartościami.

1.2.4 Wnioski

Metoda iteracyjna obliczania liczby *eta* jest poprawna, ponieważ wyniki uzyskane za jej pomocą pokrywają się z prawidłowymi wartościami. Liczba *eta* jest najmniejszą możliwą do zapisania liczbą dodatnią. Pokazuje to jej zapis bitowy. W arytmetyce `Float64`:

```
0 00000000000 0000000000000000000000000000000000000000000000000001.
```

Dziwić może fakt, że wszystkie bity cechy są zerami, oznacza to, że jest to liczba zdenormalizowana (subnormal).

1.3 MAX

1.3.1 Opis problemu

Iteracyjne wyznaczenie liczby *MAX* i porównanie jej z wartościami zwracanymi przez funkcję *realmax* z języka Julia oraz z danymi zawartymi w pliku nagłówkowym `float.h` języka C.

1.3.2 Rozwiązanie

W celu wyznaczenia liczby MAX zastosowano następujący algorytm:

$max \leftarrow 1$
while $max \cdot 2 < \infty$ **do**
    $max \leftarrow max \cdot 2$
end while
$max \leftarrow max \cdot (2 - macheps)$

1.3.3 Wyniki

Dla poszczególnych typów zmiennopozycyjnych otrzymano następujące wyniki (Tabela 3).

	max	realmax(typ)	float.h
Float16	$6.55 \cdot 10^{4}$	$6.55 \cdot 10^{4}$	—
Float32	$3.40282347 \cdot 10^{38}$	$3.40282347 \cdot 10^{38}$	$3.40282347 \cdot 10^{38}$
Float64	$1.797693134862316 \cdot 10^{308}$	$1.797693134862316 \cdot 10^{308}$	$1.797693134862316 \cdot 10^{308}$

Tabela 3: Zestawienie obliczonego MAX dla różnych typów wraz z poprawnymi wartościami.

1.3.4 Wnioski

Metoda iteracyjna obliczania liczby MAX jest poprawna, ponieważ wyniki uzyskane za jej pomocą pokrywają się z prawidłowymi wartościami.

2 Epsilon maszynowy Kahana

Kahan stwierdził, że epsilon maszynowy można uzyskać za pomocą wyrażenia $3(4/3 - 1) - 1$ w arytmetyce zmiennopozycyjnej.

2.1 Opis problemu

Eksperymentalne sprawdzenie w języku Julia słuszności stwierdzenia Kahana dla wszystkich typów zmiennopozycyjnych.

2.2 Rozwiązanie

Obliczono wartość wyrażenia podanego przez Kahana dla wszystkich typów zmiennopozycyjnych wykorzystując odpowiednie rzutowania.

2.3 Wyniki

Dla poszczególnych typów zmiennopozycyjnych otrzymano następujące wyniki (Tabela 4).

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

2.4 Wnioski

Wyniki uzyskane za pomocą obliczeń zgadzają się co do wartości z poprawnymi epsilonami maszynowymi. Dla typów `Float16` i `Float64` różnią się jednak co do znaku (poprawna byłaby ich wartość bezwzględna). Intuicje Kahana co do wyliczania epsilonu maszynowego są jednak w gruncie rzeczy poprawne. Korzysta on z faktu, że liczby 4/3 nie da się przedstawić dokładnie w systemie dwójkowym i celowo stosuje niedokładność zaokrąglenia do wyliczenia *macheps*. W dwóch przypadkach 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.

3 Rozmieszczenie liczb zmiennopozycyjnych

3.1 Opis problemu

W zadaniu należy eksperymentalnie sprawdzić w języku Julia, że w arytmetyce `Float64` liczby zmiennopozycyjne są równomiernie rozmieszczone w przedziale $[1, 2]$, a także ich rozmieszczenie w przedziałach $[\frac{1}{2}, 1]$ i $[2, 4]$.

3.2 Rozwiązanie

W celu rozwiązania zadania użyto algorytmu wyświetlającego liczby z zadanego przedziału powiększane o wskazaną wartość δ w zadanej liczbie kroków k ($x = 1 + k\delta$, gdzie $k = 1, 2, \dots$).

$liczba \leftarrow start$
for $i \leftarrow 1$ **to** $iteracje$ **do**
  $liczba \leftarrow liczba + \delta$
  wydrukuj *liczba* w prezentacji bitowej
end for

3.3 Wyniki

Poniżej (Tabela 5) przedstawiono rezultaty działania programu dla odpowiednio wybranych wartości δ, 2^{-52} dla $[1, 2]$, 2^{-53} dla $[\frac{1}{2}, 1]$ i 2^{-52} dla $[2, 4]$. Wybranych zostało kilka pierwszych i ostatnich liczb dla każdego przedziału. Z prezentacji bitowej wynika że są to kolejne liczby w arytmetyce `Float64`, ponieważ mają one tę samą cechę, a mantysa każdej kolejnej liczby (powiększonej o δ) jest większa o 1.

$[1, 2]$ $\delta = 2^{-52}$
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
⋮
0011111111111111111111111111111111111111111111111111111111111100
0011111111111111111111111111111111111111111111111111111111111101
0011111111111111111111111111111111111111111111111111111111111110
0011111111111111111111111111111111111111111111111111111111111111
$[\frac{1}{2}, 1]$ $\delta = 2^{-53}$
0011111111100000000000000000000000000000000000000000000000000001
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000011

0011111111110000000000000000000000000000000000000000000000000100
⋮
0011111111101111111111111111111111111111111111111111111111111100
0011111111101111111111111111111111111111111111111111111111111101
0011111111101111111111111111111111111111111111111111111111111110
0011111111101111111111111111111111111111111111111111111111111111
$[2, 4]$ $\delta = 2^{-51}$
0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
⋮
0100000000001111111111111111111111111111111111111111111111111100
0100000000001111111111111111111111111111111111111111111111111101
0100000000001111111111111111111111111111111111111111111111111110
0100000000001111111111111111111111111111111111111111111111111111

Tabela 5: Prezentacja bitowa wybranych liczb w danych przedziałach.

3.4 Wnioski

Z analizy danych w tabeli wynika, że między kolejnymi potęgami dwójki liczby zmiennopozycyjne są równomiernie rozmieszczone, jednak im dalej od zera tym większa jest odległość pomiędzy nimi (δ). Patrząc na prezentację bitową można łatwo zauważyć, że liczby między kolejnymi potęgami dwójki posiadają tę samą cechę, a zmianie ulega tylko mantysa. Liczb pomiędzy kolejnymi potęgami dwójki jest więc tyle samo (2^m), a co za tym idzie wraz ze zwiększaniem się przedziału maleje ich gęstość.

4 Nieodwracalność dzielenia

4.1 Opis problemu

Znalezienie najmniejszej liczby x z przedziału $(1, 2)$ w arytmetyce `Float64` takiej, że $x \cdot (1/x) \neq 1$.

4.2 Rozwiązanie

W celu rozwiązania zadania dla kolejnych liczb x w arytmetyce `Float64`, zaczynając od najmniejszej liczby większej od 1, zostało sprawdzone czy warunek $x \cdot (1/x) \neq 1$ zachodzi. W momencie znalezienia pierwszej takiej liczby program przerywał pracę i wyświetlał wynik na ekranie.

4.3 Wyniki

Najmniejszą liczbą znalezioną w wyniku działania programu jest: 1.000000057228997.

4.4 Wnioski

Zadanie pokazuje, że działania arytmetyczne na liczbach zmiennopozycyjnych mogą generować błędy związane z zaokrąglaniem wyliczonych wartości. Przy używaniu typów zmiennopozycyjnych takie błędy często są nieuniknione. Należy jednak pamiętać, że pewne działania użytkownika mogą je potęgować, czego w bardzo wielu sytuacjach da się uniknąć jak np. w tym przypadku zamiast dzielenia, które w arytmetyce zmiennopozycyjnej nie zawsze jest odwracalne, wyliczyć ręcznie wartość i wpisać 1.

5 Iloczyn skalarny

5.1 Opis problemu

Obliczenie iloczynu skalarnego danych wektorów z wykorzystaniem czterech różnych algorytmów sumowania dla typów `Float32` i `Float64`.
$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$
$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$

5.2 Rozwiązanie

W programie zaimplementowano podane algorytmy:

1. "w przód": $\sum_{i=1}^{n} x_i y_i$;
2. "w tył": $\sum_{i=n}^{1} x_i y_i$;
3. 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; zostało to wykonane za pomocą sortowania i odpowiedniego dodania elementów tablicy sum częściowych;
4. metoda przeciwna do sposobu *3*.

5.3 Wyniki

Otrzymano następujące wyniki (Tabela 6): Prawidłowy iloczyn skalarny danych w zadaniu wek-

	1	2	3	4
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 6: Iloczyn skalarny danych wektorów.

torów to $-1.00657107000000 \cdot 10^{-11}$. Wszystkie otrzymane wyniki są od niego różne.

5.4 Wnioski

Zadanie pokazuje, że kolejność wykonywania działań nie jest bez znaczenia. Na przykład dodanie do bardzo dużej liczby liczby w stosunku do niej bardzo małej generuje błędy. Można zauważyć także, że im więcej wykonywanych jest działań na liczbach zmiennopozycyjnych tym większy jest błąd względny. Jednym ze sposobów na uniknięcie dużych błędów, kiedy inne metody zawodzą, jest użycie arytmetyki o większej precyzji. Użycie `Float64` zamiast `Float32` w zadaniu w znaczący sposób przybliżyło uzyskane wyniki do poprawnego, jednak nawet to nie dało zadowalających rezultatów. Wektory dane w zadaniu są prawie ortogonalne, co sprawia że liczenie ich iloczynu skalarnego prowadzi do dużych błędów względnych.

6 Przybliżenie funkcji

6.1 Opis problemu

Obliczenie w arytmetyce `Float64` wartości funkcji $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla kolejnych wartości $x = 8^{-1}, 8^{-2}, 8^{-3} \ldots$.

6.2 Rozwiązanie

Obliczone zostały wartości funkcji f i g dla kolejnych argumentów x w pętli.

6.3 Wyniki

Przykładowe wyniki działania programu przedstawiono poniżej (Tabela 7). Funkcje f i g są sobie równe i dla kilku początkowych argumentów ich wartości są rzeczywiście zbliżone. Jednak funkcja f bardzo szybko (dla 8−9) osiągnęła wartość 0.0. Funkcja g jeszcze dla $x = 8^{-178}$ pokazuje wartość różną od zera ($1.6 \cdot 10^{-322}$).

x	f	g
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	$1.9073468138230965 \cdot 10^{-6}$	$1.907346813826566 \cdot 10^{-6}$
$\vdots$	$\vdots$	$\vdots$
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^{-177}	0.0	$1.012 \cdot 10^{-320}$
8^{-178}	0.0	$1.6 \cdot 10^{-322}$
8^{-179}	0.0	0.0

Tabela 7: Wartości funkcji f i g dla kolejnych argumentów.

6.4 Wnioski

Dane w zadaniu funkcje dla $x \to 0$ dążą do zera, teoretycznie nigdy nie powinny tego zera osiągnąć, oczywiście arytmetyka komputera nie jest na tyle dokładna żeby na to pozwolić. Funkcja f jednak osiąga wartość zero dla stosunkowo dużych x. Dzieje się tak, ponieważ odejmowane są bardzo bliskie sobie liczby, co powoduje utratę cyfr znaczących. Funkcja g nie generuje takiego błędu i jak wynika z analizy danych jest dużo bardziej dokładna niż funkcja f, a jej błąd wynika w zasadzie z niedokładności arytmetyki. Odejmowanie bliskich sobie licz jest niebezpieczne, gdyż zawsze generuje utratę cyfr znaczących i należy w miarę możliwości go unikać. W tym celu można zastosować wyrażenie w alternatywnej postaci lub w szczególności, kiedy jest to niemożliwe zastosować większą precyzję.

7 Przybliżenie pochodnej

7.1 Opis problemu

Obliczenie zadanym wzorem w arytmetyce `Float64` przybliżonej wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ oraz błędów $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$ $(n = 0, 1, 2, \ldots, 54)$.

7.2 Rozwiązanie

W celu obliczenia przybliżonej wartości pochodnej funkcji zastosowano podany wzór $\tilde{f}'(x_0) = \frac{f(x_0+h)-f(x_0)}{h}$. Wyprowadzony został także wzór na pochodną funkcji $f'(x) = \cos(x) - 3 \cdot \sin(3x)$ w celu umożliwienia obliczenia błędu. Dla kolejnych wartości h w pętli została obliczona przybliżona pochodna, błąd bezwzględny, a także wartość $h + 1$.

7.3 Wyniki

Wyniki obliczeń dla poszczególnych wartości h przedstawiono poniżej (Tabela 8). 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.

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 8: Wartości funkcji f i g dla kolejnych argumentów.

7.4 Wnioski

Analiza wyników wyrażenia $1 + h$ pokazuje, że w pewnym momencie obliczania przybliżonej pochodnej h stało się na tyle małe w stosunku do 1, że 1 niejako "pochłonęło" h. Najlepiej widać to dla najmniejszych wartości h, gdzie wynik operacji $1 + h = 1$. Oczywiście, taki wynik zaburza poprawność działania funkcji przybliżenia pochodnej. Uzyskane wyniki pokazują, że należy unikać dodawania do siebie liczb które znacznie różnią się wykładnikami, bo powoduje to błędy, które potem mogą być potęgowane przez dalsze obliczenia. Drugą rzeczą, która mogła mieć wpływ na zmniejszenie dokładności przybliżenia jest odejmowanie bliskich sobie wartości $f(x_0 + h)$ i $f(x_0)$ (funkcja nie rośnie szybko) szczególnie dla bardzo małych h. Jest to związane z utratą cyfr znaczących i także może zaburzać wynik obliczeń.