Obliczenia Naukowe - sprawozdanie nr 1

Paweł Wilkosz

24 października 2022

1 Parametry arytmetyki zmiennopozycyjnej

1.1 Epsilon maszynowy

Epsilon maszynowy to jeden z najwazniejszych parametrów arytmetyki zmiennopozycyjnej. Można go zdefiniować jako najmniejsze takie x, że $1 + x > 1$. W poniższej tabeli przedstawione są wartości epsilona maszynowego zwracane przez funkcję `eps` ze standardowej biblioteki języka Julia.

Typ	`eps`
Float16	0.000977
Float32	1.1920929e-7
Float64	2.220446049250313e-16

Parametr ten można również obliczyć iteracyjnie, za pomocą poniższej procedury.

```
function itereps(type)
  current_num = one(type)
  while (one(type) + current_num / 2) > one(type)
    current_num /= 2
  end
  return current_num
end
```

Uruchomienie powyższej funkcji daje wyniki zgodne z rezultatami funkcji `eps`.

Typ	`itereps`
Float16	0.000977
Float32	1.1920929e-7
Float64	2.220446049250313e-16

1.2 Liczba Eta

Liczba Eta, równa wspomnianej na wykładzie liczby MIN_{sub} to najmniejsza (co do wartości bezwzględnej) nieznormalizowana liczba reprezentowalna w danej arytmetyce.

Typ	`itereta`
Float16	6.0e-8
Float32	1.0e-45
Float64	5.0e-324

1.3 Float min i Float max

Liczby `floatmin` i `floatmax` opisują odpowiednio, minimalne i maksymalne znormalizowane liczby dodatnie możliwe do przedstawienia w danej arytmetyce. Poniższa tabela przedstawia wartości zwracane przez funkcję `floatmin` dla wszystkich typów zmiennoprzecinkowych w języku Julia.

Typ	`floatmin`
Float16	6.104e-5
Float32	1.1754944e-38
Float64	2.2250738585072014e-308

Liczbę `floatmax` możemy obliczyć zarówno przy pomocy wbudowanej funkcji języka Julia jak i iteracyjnie, przy pomocy poniższej metody.

```
function itermax(type)
 current_num = prevfloat(one(type))
 while !isinf(current_num * 2)
  current_num *= 2
 end
 return current_num
end
```

Obliczona liczba jest efektem kolejnego mnożenie przez dwa liczby `prevfloat(one(type))`. Wybór początkowej liczby podyktowany jest tym, że ma ona maksymalną mantysę, wystarczy więc zwiększyć cechę - mnożąc przez dwa.

W poniższej tabeli przedstawione są wartości zwracane przez funkcję `itermax` porównane z wynikami funkcji `floatmax`.

Typ	`itermax`	`floatmax`
Float16	6.55e4	6.55e4
Float32	3.4028235e38	3.4028235e38
Float64	1.7976931348623157e308	1.7976931348623157e308

Równość wyników `itermax` i `floatmax` dowodzi poprawności wyżej opisanego algorytmu.

1.4 Podsumowanie

Arytmetyka IEEE-754 niesie za sobą wiele ograniczeń. Możemy przekonać się o nich eksperymentalnie, używając wbudowanych funkcji języka Julia, bądź samodzielnie sprawdzając implementację. Zajrzywszy do pliku `float.h` kompilatora `gcc` przekonamy się, że zawarte tam wartości są zgodne z tymi, które wynikają z powyższych eksperymentów.

2 Epsilon maszynowy metodą Kahana

Do obliczenia epsilona maszynowego posłużyć może również wyrażenie:

$$3(\frac{4}{3} - 1) - 1$$

Implementując je w języku Julia i uruchamiając kolejno dla odpowiednich arytmetyk otrzymujemy:

Typ	Obliczony epsilon
Float16	-0.000977
Float32	1.1920929e-7
Float64	-2.220446049250313e-16

Jak łatwo zaobserwować, metoda ta zwraca idealną wartość epsilona maszynowego z dokładnością co do wartości bezwzględnej. Dla typów `Float16` i `Float64` wartości są ujemne.

Takie wyniki spowodowane są zaokrągleniem liczby $\frac{4}{3}$ w pamięci komputera. Liczba ta w systemie binarnym ma nieskończone rozwinięcie więc w zależności od parzystości długości mantysy, faktyczna wartość w pamięci jest większa bądź mniejsza o `macheps` od porządanej przez nas wartości.

3 Rozmieszczenie liczb zmiennoprzecinkowych

Celem zadania jest sprawdzenie czy liczby z zakresu $[1, 2]$ są rozmieszczone rónomiernie z krokiem $\delta = 2^{-52}$.

W celu weryfikacji tego faktu, można użyć funkcji `bitstring`. Jeżeli postawiona teza jest prawdziwa, dodanie do kolejnych liczb z zakresu liczby δ powinno powodować zwiększenie mantysy o 1.

```
bitstring(1+delta)
0011111111110000000000000000000000000000000000000000000000000001
bitstring(1+2*delta)
0011111111110000000000000000000000000000000000000000000000000011
bitstring(1+3*delta)
0011111111110000000000000000000000000000000000000000000000000100
```

Wykonanie obliczeń dla pierwszych trzech liczb z zakresu potwierdza tezę.

Własność tę można uogólnić dla wszystkich zakresów liczb ze stałą cechą, to jest postaci $[2^k, 2^{k+1}]$ gdzie $k \in \mathbb{Z}$. Jako że liczba możliwych mantys jest stała, w każdym z takich zakresów znajduje się tyle samo liczb. Można przetestować tę własność na zakresach $[\frac{1}{2}, 1]$ oraz $[2, 4]$. W związku z tym, że zakresy te są odpowiednio dwa razy mniejsze lub dwa razy większe od początkowo badanego, krok pomiędzy poszczególnymi liczbami również musi ulec zmianie.

Dla liczb z zakresu $[2, 4]$ krok wynosi $\delta = 2^{-51}$

```
bitstring(2+delta)
0100000000000000000000000000000000000000000000000000000000000001
bitstring(2+2*delta)
0100000000000000000000000000000000000000000000000000000000000011
bitstring(2+3*delta)
0100000000000000000000000000000000000000000000000000000000000100
```

a dla liczb z zakresu $[\frac{1}{2}, 1]$ krok wynosi $\delta = 2^{-53}$

```
bitstring(0.5+delta)
0011111111100000000000000000000000000000000000000000000000000001
bitstring(0.5+2*delta)
0011111111100000000000000000000000000000000000000000000000000011
bitstring(0.5+3*delta)
0011111111100000000000000000000000000000000000000000000000000100
```

Z powyższych doświadczeń wynika, iż różnica pomiędzy kolejnymi liczbami zmiennoprzecinkowymi rośnie wraz z oddalaniem się od zera. Co za tym idzie, im większa cecha, tym mniejsza bezwzględna dokładność obliczeń.

4 Odwracalność liczb

Zgodnie ze standardową matematyką, wyrażenie $x \cdot \frac{1}{x} = 1$ jest zawsze prawdziwe dla $x \neq 0$. Niestety w arytmetyka zmiennoprzecinkowa sprawia, że równanie to nie jest prawdziwe dla wszystkich liczb.

Aby znaleźć najmniejszą liczbę niespełniającą tego warunku w zakresie $[1.0, 2.0]$ możemy użyć poniższej procedury.

```
function find_irreversible()
  number = one(Float64)
  while isreversible(number) && number < 2.0
    number = nextfloat(number)
  end
  return number
end
```

Wynikiem jej działania jest liczba `1.000000057228997`.

Istnienie takiej liczby jest spowodowane ograniczonym systemem reprezentacji liczb w pamięci. Wiele liczb rzeczywistych przedstawialnych w systemie dziesiętnym ze skończonym rozwinięciem, w systemie binarnym posiada rozwinięcie nieskończone. Powoduje to duże utraty dokładności podczas dzielenia niecałkowitoliczbowego.

5 Iloczyn skalarny dwóch wektorów

Zadanie bada problem liczenia iloczynu skalarnego dwóch wektorów. Porównujemy cztery algorytmy służące do tego celu.

1. suma iloczynów kolejnych współrzędnych wykonana od początku tablicy do końca `fronttoback`
2. suma iloczynów kolejnych współrzędnych wykonana w odwrotnej kolejności `backtofront`
3. suma iloczynów kolejnych współrzednych wykonana w kolejności od wyników najdalszych zeru do najbliższych, z podziałem na dodatnie i ujemne `bigtosmall`
4. suma iloczynów kolejnych współrzednych wykonana w kolejności od wyników najbliższych zeru do najdalszych, z podziałem na dodatnie i ujemne `smalltobig`

W poniższej tabeli przedstawiono wyniki eksperymentu dla danych

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

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

Float32	`fronttoback`	-0.2499443
	`backtofront`	-0.2043457
	`bigtosmall`	-0.25
	`smalltobig`	-0.25
Float64	`frontoback`	1.0251881368296672e-10
	`backtofront`	-1.5643308870494366e-10
	`bigtosmall`	0.0
	`smalltobig`	0.0

Porównując te wyniki z poprawnym wynikiem wynoszącym $1.02519 * 10^{-10}$, najlepsze wyniki zagwarantowała arytmetyka `Float64` wraz z metodą `fronttoback`. Metoda `backtofront` zwróciła wyniki poprawnego rzędu jednak z istotnym błędem i przeciwnym znakiem. Metody `bigtosmall` i `smalltobig` zwróciły niepoprawne wyniki. W arytmetyce `Float32` wszystkie wyniki są niepoprawne ze względu na utratę dokładności przy zrównywaniu cech przy dodawaniu liczb różnego rzędu

Eksperyment ten ujawnia niecodzienną własność dodawania w arytmetyce IEEE-754, mianowicie brak przemienności.

6 Obliczanie wartości przekształconych funkcji

Zadanie polegało na obliczeniu wartości dwóch równoważnych funkcji.

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

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

W poniższej tabeli zaprezentowane jest porównanie wyników obu funkcji dla wartości x ze zbioru $\{8^{-i} : i \in [10]\}$

x	$f(x)$	$g(x)$
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

Można zaobserwować, że pomimo matematycznej równości pomiędzy funkcjami, obliczenia wykonane w języku Julia zwracają różne wyniki. Dla większych wartości, różnice są niewielkie, jednak gdy argumenty funkcji są odpowiednio bliskie 0, wartości funkcji f wynoszą 0. Wartości funkcji g natomiast wciąż zwraca niezerowe liczby. Na tej podstawie można wnioskować wyższą dokładność drugiej z badanych funkcji. Może to być spowodowane małą dokładnością odejmowania bliskich sobie liczb, którego unikamy stosując funkcję g.

7 Obliczanie przybliżonej wartości pochodnej

Zadanie polegało na obliczeniu przybliżonej wartości pochodnej przy pomocy wzoru:

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

gdzie przez h oznaczamy dokładność obliczeń.

Badaną funkcją była $f(x) = \sin x + \cos 3x$, której pochodna dana jest wzorem $f'(x) = \cos x - 3\sin 3x$. Poniższa tabela przedstawia przybliżone wartości pochodnej oraz błąd obliczeń w punkcie $x_0 = 1$ dla dokładności $h \in \{2^{-i} : i \in [54]\}$.

h	$\widetilde{f}'(x)$	$\lvert\widetilde{f}'(x) - f'(x)\rvert$
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

Jak można zaobserwować, wraz ze spadkiem h dokładność przybliżenia wartości pochodnej jest większa do pewnej waroości. Błąd obliczeń jest minimalny dla $h = 2^{-28}$ a dla mniejszych wartości h rośnie. Wzrost błędu obliczeń można wytłumaczyć tym, że w arytmetyce IEEE-754 dla bardzo małych h, zachodzi $1 + h = 1$.

8 Podsumowanie

Arytmetyka IEEE-754 pozwala na dokonywanie efektywne zapiwywanie liczb w pamięci komputera i dokonywanie na nich obliczeń. Jej dokładność jest jednak ograniczona, o czym należy pamiętać wykonując obliczenia wymagającej dużej precyzji. W szczególności na błąd jesteśmy narażeni wykonując obliczenia na liczbach różnego rzędu, to jest bardzo bliskich i bardzo dalekich od zera.