Obliczenia Nukowe - Laboratoria - Lista 1

Jakub Jaśków

October 22, 2023

1 Zad

1.1 Epsilon Maszynowy

Opis i cel

Wyznaczenie w sposób iteracyjny wartości machine epsilon (zera maszynowego) dla arytmetyki Float16, Float32 i Float64. Porównanie otrzymanych wartości z funkcją **eps()** języka Julia oraz wartościami znajdującymi się w pliku nagłówkowym **float.h** języka C.
Jaki związek ma liczba macheps z precyzją arytmetyki(eta)?

Rozwiązanie

Epsilon maszynowy to najmniejsza liczba taka, że machEps $+ 1 > 1$. Aby wyznaczyć zero maszynowe zaczniemy od 2 i stopniowo będziemy mnożyć ją przez 0.5. Jest to porównywalne do przesuwania bitów w prawą stronę.

Wyniki

Wyniki zwracane przez **eps()** pokrywają się z tymi wyznaczonymi iteracyjnie. Nie są też odległe od wartości które można znaleźć w **float.h**. Float16 nie ma tu dopowiednika.

	iterative	**eps()**	**float.h**
Float16	0.000977	0.000977	
Float32	1.1920929e-7	1.1920929e-7	1.19209e-07
Float64	2.220446049250313e-16	2.220446049250313e-16	2.22045e-16

Związek *macheps* z *eta*

Jaki jest zatem związek zera maszynowego z precyzją arytmetyki? Liczbę eta wyznaczoną wzorem

$$\eta = 0.5 * \beta^{1-t}$$

gdzie β jest bazą rozwinięcia (tutaj 2), a t - liczba cyfr mantysy znormalizowanej do przedziału $[\frac{1}{\beta}, 1]$. Porównując uzyskane wyniki z danymi z przedstawionymi na wykładzie można wywnioskować, że:

$$macheps = 2eta$$

1.2 Precyzja arytmetyki

Opis i cel

Wyznaczenie w sposób iteracyjny wartości eta (precyzja arytmetyki) dla arytmetyki Float16, Float32 i Float64. Porównanie otrzymanych wartości z funkcją **nextfloat(type(0))** języka Julia.
Jaki związek ma *eta* z MIN_{sub}?

Rozwiązanie

η to najmniejsza liczba > 0. Metoda wyznaczenia *eta* jest analogiczna do wyznaczania macheps - zaczynamy od 2 i mnożymy je razy 0.5 tak długo do póki nie wyjdziemy z pętli while.

Wyniki

Jak widzimy również i w tym przypadku wyniki wyznaczone w sposób iteratywny pokrywają się z wartościami uzyskanymi dzięki funckją bibliotecznym.

	iterative	**nextfloat(type(0))**
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	2.220446049250313e-16	2.220446049250313e-16

Związek *eta* z MIN_{sub}

Liczba eta - najmniejsza liczba $>$ 0 możliwa do zapisania w danej arytmetyce fl. Jest to najmniejsza liczba zdenormalizowana; jej wszystkie cechy są wyzerowane a ostatni bit mantysy to 1.
Więc $eta = MIN_{sub}$.

1.3 MAX

Opis i cel

Wyznaczenie w sposób iteracyjny wartości MAX (największej możliwej do wyrażenia liczba) dla arytmetyki Float16, Float32 i Float64. Porównanie otrzymanych wartości z funkcją **floatmax(type(0))** języka Julia oraz odpowiadającym im wartościami znajdującymi się w pliku **float.h**.

Co zwracają funkcje bibliotekowe **floatmin(Float32)** i **floatmin(Float64)**, i jaki jest ich związek z liczbą MIN_{nor}?

Rozwiązanie

Liczbą maksymalną będzie liczba posiadająca mantysę składającą się z samych jedynek oraz największą dopuszczalną cechą. Generujemy pierwszą liczbę x funckją **prevfloat(1.0)** a nastpęnie mnożymy x przez 2 w pętli while - sprawdzając tym samym czy **isinf**(x) = **false**.

Wyniki

	iterative	**maxfloat(type)**	**float.h**
Float16	6.55e4	6.55e4	
Float32	3.4028235e38	3.4028235e38	3.40282e38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.79769e308

Związek *floatmin()* z MIN_{nor}

$\mathbf{MIN_{nor}}$ to najmniejsza liczba znormalizowana reprezentowana w danej arytmetyce pozycyjnej.

	floatmin()	$\mathbf{MIN_{nor}}$
Float32	1.1754944e-38	1.2e-38
Float64	2.2250738585072014e-308	2.2e-308

Wartości **floatmin()** są zbliżone do wartości $\mathbf{MIN_{nor}}$ podanych na wykładzie.

2 Zad

Opis i cel

Sprawdź eksperymentalnie, czy:

$$3(4/3 - 1) - 1 = macheps$$

w danej arytmetyce pozycyjnej. Arytmetyki do sprawdzenia: Float16, Float32, Float64.

Rozwiązanie

```
floatTypes = [Float16, Float32, Float64]
println("Zadanie 2")
for type in floatTypes
    println("\n$type(3(4/3-1) - 1) = ", type(3.0*type(type(4.0 / 3.0) - 1.0) - 1))
    println("macheps($type) = ", eps(type))
end
```

Wyniki

	3(4/3-1)-1	**eps()**
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Wnioski

Biorąc pod uwagę fakt, że wyniki pokrywają się co do wartości bezwzględnych można stwierdzić, że wzór Kahana jest prawidłowy.

3 Zad

Opis i cel

Sprawdź eksperymentalnie, czy w arytmetyce Float64 w przedziale $[1, 2]$ liczby są równomiernie rozłożone z krokiem $\delta = 2^{-52}$. Czyli każda liczba w tej arytmetyce może być przedstawiona jako $x = 1 + k\delta$, gdzie $k = 1, 2, ..., 2^{52} - 1$.
Sprawdź δ dla $x \in [\frac{1}{2}, 1]$ oraz $x \in [2, 4]$.

Rozwiązanie

Aby wykonać to zadanie użyjemy funkcji **bitstring()** języka Julia, aby wypisać kolejne liczby.

Wyniki

$[1, 2]$, $\delta = 2^{-52}$
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101

$[\frac{1}{2}, 1]$, $\delta = 2^{-53}$
0011111111100000000000000000000000000000000000000000000000000001
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000011
0011111111100000000000000000000000000000000000000000000000000100
0011111111100000000000000000000000000000000000000000000000000101

$[2, 4]$, $\delta = 2^{-51}$
0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
0100000000000000000000000000000000000000000000000000000000000101

Analizując pierwsze 5 liczb danego przedziału możemy stwierdzić, że liczby te różnią się o jeden bit, więc iterujemy przez wszystkie liczby w danym przedziale. Z tąd możemy wnioskować, że dla danego przedziału:

$$x = start + k * \delta_i$$

.

Wnioski

Wyniki eksperymentu dowodzą, że liczby posiadające taką samą cechą są rozmieszczone regularnie. Np. dla przedziału: $[8, 16]$ $\delta = 2^{-49}$

4 Zad

Opis i cel

Znajdź eksperymentalnie w arytmetyce Float64 najmniejszą taką liczbę, że:

$$fl(xfl(1/x)) \neq 1$$

, gdzie $x \in [1, 2]$

Rozwiązanie

Rozwiązanie jest proste. Wystarczy w pętli while czy podana powyżej zależność jest prawdziwa. Jeżeli tak: $x = nextfloat(x)$

Wyniki

$$minval = 1.0000000572289969$$

5 Zad

Opis i cel

Napisz program w języku Julia realizujący następujący eksperyment obliczania iloczynu skalarnego dwóch wektorów:
$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$
$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$.
Zaimplementuj poniższe algorytmy i policz sumę na cztery sposoby dla n = 5:
(a) "w przód" $\sum_{i=1}^{n} x_i * y_i$, tj. algorytm:

```
S := 0
for i := 1 to n do
S := S + xi *yi
end for
```

(b) "w tył" $\sum_{i=n}^{1} x_i * y_i$, tj. algorytm:

```
S := 0
for i := n downto 1 do
S := S + xi *yi
end for
```

(c) od największego do najmniejszego (dodaj dodatnie liczby w porządku od największego do najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego, a następnie daj do siebie obliczone sumy częściowe)
(d) od najmniejszego do największego (przeciwnie do metody (c)). Użyj pojedynczej i podwójnej precyzji (typy Float32 i Float64 w języku Julia). Porównaj wyniki z prawidłową wartością (dokładność do 15 cyfr) $-1.00657107000000 * 10 - 11$.

Rozwiązanie

Podpunkty a) i b) to formalność. Wystarczy przepisać algorytm do Julii. W podpunkcie c) i d) wystarczy rozdzielić liczby ujemne i dodatnie a następnie wykonać dodawanie w sposób odpowiedni dla każdego z podpunktów.

Wyniki

algorytm	**Float32**	**Float64**
w przód	-0.4999443	1.0251881368296672e-10
w tył	-0.4543457	-1.5643308870494366e-10
malejąco	-0.5	0.0
rosnąco	-0.5	0.0

5.1 Wnioski

Jak widać zwiększenie precyzji poskutkowało zbliżeniem do odpowiedniej wartości, natomiast żaden z algorytmów nie zwrócił nam poprawnej liczby. Można więc wnioskować, że kolejność wykonywania działań na liczbach jest w stanie znacznie wpłynąć na wynik.

6 Zad

Opis

Policz w języku Julia w arytmetyce Float64

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

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

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

Rozwiązanie

Wyliczenie wartości funkcji poprzez funkcje bibliotekowe języka Julia z zachowaniem arytemtyki Float64.

Wyniki

x	$f(x)$	$g(x)$
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	$1.9073468138230965 \times 10^{-6}$	$1.907346813826566 \times 10^{-6}$
8^{-4}	$2.9802321943606103 \times 10^{-8}$	$2.9802321943606116 \times 10^{-8}$
8^{-5}	$4.656612873077393 \times 10^{-10}$	$4.6566128719931904 \times 10^{-10}$
8^{-6}	$7.275957614183426 \times 10^{-12}$	$7.275957614156956 \times 10^{-12}$
8^{-7}	$1.1368683772161603 \times 10^{-13}$	$1.1368683772160957 \times 10^{-13}$
8^{-8}	$1.7763568394002505 \times 10^{-15}$	$1.7763568394002489 \times 10^{-15}$
8^{-9}	0.0	$2.7755575615628914 \times 10^{-17}$
8^{-10}	0.0	$4.336808689942018 \times 10^{-19}$
8^{-11}	0.0	$6.776263578034403 \times 10^{-21}$
8^{-12}	0.0	$1.0587911840678754 \times 10^{-22}$
8^{-13}	0.0	$1.6543612251060553 \times 10^{-24}$
8^{-14}	0.0	$2.5849394142282115 \times 10^{-26}$
8^{-15}	0.0	$4.0389678347315804 \times 10^{-28}$
8^{-16}	0.0	$6.310887241768095 \times 10^{-30}$
8^{-17}	0.0	$9.860761315262648 \times 10^{-32}$
8^{-18}	0.0	$1.5407439555097887 \times 10^{-33}$
8^{-19}	0.0	$2.407412430484045 \times 10^{-35}$
8^{-20}	0.0	$3.76158192263132 \times 10^{-37}$

Wnioski

Uzyskane wyniki różnią się od siebie pomimo, że $f = g$. Dla większych wyników wartości zwracane przez funkcje są podobne, ale od $x = 8^{-8}$ funkcja f zaczyna zwracać 0. Zachowanie to spodwodowane jest odejmowaniem przez funkcję f wartości zbliżonych do siebie przez co tracimy cyfry znaczące i uzyskujemy błędny wynik. W funkcji g unikamy odejmowania bliskich siebie wartości co czyni tą funkcję dokładniejszą.

7 Zad

Opis i cel

Skorzystaj ze wzoru na przybliżoną wartość funkcji:

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

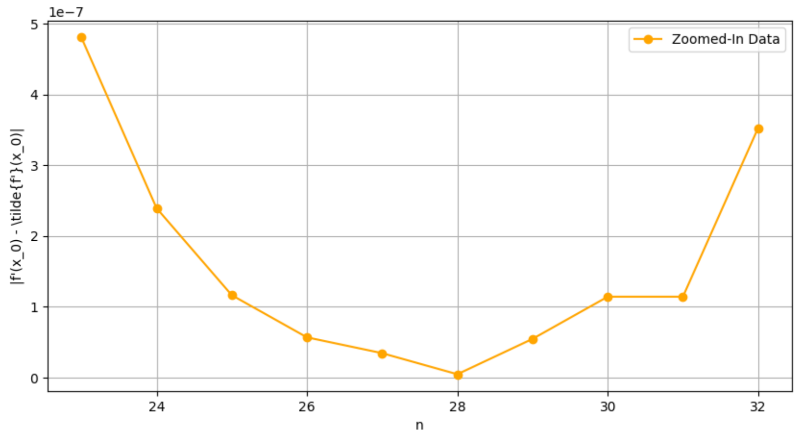
oraz błąd $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$, gdzie $n \in [1, 2, 3, ..., 54]$.
Jak wytłumaczyć, że od pewnego momentu zmniejszanie wartości h nie poprawia przybliżenia wartości pochodnej?

Rozwiązanie

Prawdziwa pochodna funkcji $f(x) = cos(x) - 3sin(3x)$. Obliczenie błędu $|f'(x_0) - \tilde{f}'(x_0)|$ dla każdej wartości $h = 2^{-n}$, gdzie $n \in [1, 2, 3, ..., 54]$.

Wyniki

h	$f'(x_0)$	$\lvert f'(x_0) - \tilde{f}'(x_0)\rvert$	$1 + h$
2.0^{-0}	2.0179892252685967	1.9010469435800585	2.0
2.0^{-1}	1.8704413979316472	1.753499116243109	1.5
2.0^{-2}	1.1077870952342974	0.9908448135457593	1.25
2.0^{-3}	0.6232412792975817	0.5062989976090435	1.125
2.0^{-4}	0.3704000662035192	0.253457784514981	1.0625
.	.	.	.
.	.	.	.
.	.	.	.
2.0^{-26}	0.11694233864545822	5.6956920069239914e-8	1.0000000149011612
2.0^{-27}	0.11694231629371643	3.460517827846843e-8	1.0000000074505806
2.0^{-28}	0.11694228649139404	4.802855890773117e-9	1.0000000037252903
2.0^{-29}	0.11694222688674927	5.480178888461751e-8	1.0000000018626451
2.0^{-30}	0.11694216728210449	1.1440643366000813e-7	1.0000000009313226
.	.	.	.
.	.	.	.
.	.	.	.
2.0^{-49}	0.125	0.008057718311461848	1.0000000000000018
2.0^{-50}	0.0	0.11694228168853815	1.0000000000000009
2.0^{-51}	0.0	0.11694228168853815	1.0000000000000004
2.0^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2.0^{-53}	0.0	0.11694228168853815	1.0
2.0^{-54}	0.0	0.11694228168853815	1.0



Wnioski

Możemy zauważyć, że dla $n = 28$ uzyskujemy najbliższy wynik równy prawdziwemu. Zmniejszenie h od tej wartości nie poprawia dokładności wyniku. Błąd zmniejsza się dla $n < 28$ a dla $n >= 29$ rośnie. Dzieje się tak na skutek dodawania małych wartości liczbowych do dużych, skutkując ich zepsuciem.