Obliczenia naukowe
Lista 1

Szymon Janiak

October 25, 2024

1 Zadanie 1

1.1 Macheps

macheps to najmniejsza liczba > 0 taka, że $fl(1.0 + macheps) > 1.0$ i $fl(1.0 + macheps) = 1 + macheps$

1.1.1 Opis problemu

Wyznaczenie iteracyjnie epsilonów maszynowych dla wszystkich dostępnych typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754.

1.1.2 Rozwiązanie

Algorithm 1 *macheps* iteracyjnie

$x \leftarrow 1.0$
$macheps \leftarrow 1.0$
while $x + \frac{macheps}{2} \neq x$ **do**
  $macheps \leftarrow \frac{macheps}{2}$
end while
return *macheps*

1.1.3 Wyniki

Źródło	Float16	Float32	Float64
Mój algorytm	0.000977	1.1920929e-7	2.220446049250313e-16
eps()	0.000977	1.1920929e-7	2.220446049250313e-16
float.h	-	1.19209e-07	2.22045e-16

1.1.4 Wnioski

Wyniki obliczone iteracyjnie są zgodne z wynikami zwracanymi przez funkcję *eps* w Julii. Wyniki pokrywają się również z wartościami z pliku nagłówkowego $float.h$ w języku C, są one zgodne ze standardem IEEE-754 co potwierdza poprawność implementacji. Liczba macheps jest ściśle powiązana z precyzją arytmetyki, im wyższa precyzja tym mniejszy będzie macheps.

1.2 Eta

eta to najmniejsza liczba taka, że $eta > 0.0$

1.2.1 Opis problemu

Wyznaczenie iteracyjnie liczb maszynowych eta dla wszystkich dostępnych typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754.

1.2.2 Rozwiązanie

Algorithm 2 *eta* iteracyjnie

$eta \leftarrow 1.0$
while $\frac{eta}{2} \neq 0.0$ **do**
  $eta \leftarrow \frac{eta}{2}$
end while
return *eta*

1.2.3 Wyniki

Źródło	Float16	Float32	Float64
Mój algorytm	6.0e-8	1.0e-45	5.0e-324
nextfloat(0.0)	6.0e-8	1.0e-45	5.0e-324

1.2.4 Wnioski

Wyniki obu funkcji pokrywają się ze standardem IEEE-754 co potwierdza poprawność implementacji.

$MIN_{sub} = 2^{1-t} * 2^{c_{min}}$
gdzie t to liczba cyfr mantysy, a $c_{min} = -2^{d-1} + 2$ gdzie, d oznacza liczbę bitów przeznaczonych na zapis cechy.

	Float16	Float32	Float64
MIN_{sub}	6.0e-8	1.0e-45	5.0e-324

Z tego wynika, że $eta = MIN_{sub}$, czyli najmniejsza liczba (zdenormalizowana), którą da się reprezentować w danej arytmetyce

1.3 MIN_{nor}

$MIN_{nor} = 2^{c_{min}}$
gdzie c_{min} jest tym samym co przy MIN_{sub}

	Float16	Float32	Float64
MIN_{nor}	6.104e-5	1.1754944e-38	2.2250738585072014e-308
$floatmin()$	6.104e-5	1.1754944e-38	2.2250738585072014e-308

1.3.1 Wnioski

$floatmin()$ dla danej arytmetyki jest równy z jej MIN_{nor} z czego wynika ze funkcja ta zwraca najmniejszą liczbę znormalizowaną w danej arytmetyce

1.4 MAX

1.4.1 Opis Problemu

Wyznaczenie iteracyjnie liczbę MAX dla wszystkich typów zmiennoprzecinkowych, zgodnych ze standardem IEEE 754.

1.4.2 Rozwiązanie

Algorithm 3 MAX iteracyjnie

$max \leftarrow prevfloat(1.0$
while $max * 2 \neq \infty$ **do**
  $max \leftarrow MAX * 2$
end while
return MAX

1.4.3 Wyniki

Źródło	Float16	Float32	Float64
Mój algorytm	6.55e4	3.4028235e38	1.7976931348623157e308
floatmax()	6.55e4	3.4028235e38	1.7976931348623157e308
float.h	-	3.40282e+38	1.79769e+308

1.4.4 Wnioski

Wyniki pokrywają się również z wartościami z pliku nagłówkowego $float.h$ w języku C, są one zgodne ze standardem IEEE-754 co potwierdza poprawność implementacji.

2 Zadanie 2

2.1 Twierdzenie Khan'a

Epislon maszynowy *macheps* można otrzymać obliczając wyrażenie $3.0 * (\frac{4.0}{3.0} - 1.0) - 1.0$

2.2 Opis problemu

Sprawdzenie czy stwierdzenie Khan'a jest słuszne dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64.

2.3 Wyniki

Źródło	Float16	Float32	Float64
Khan	-0.000977	1.1920929e-7	-2.220446049250313e-16
eps()	0.000977	1.1920929e-7	2.220446049250313e-16

2.4 Wnioski

Wyniki z wyrażenia od Khan'a różnią się znakiem dla Float16 i Float64, jednakże pomijając znak otrzymujemy te same wartości i w tym przypadku jego twierdzenie jest słuszne.

3 Zadanie 3

3.1 Opis problemu

Sprawdzenie rozmieszczenia liczb zmiennoprzecinkowych Float64 w standardzie IEEE 754 w przedziałach [1, 2], $[\frac{1}{2}, 1]$ oraz [2, 4].

3.2 Rozwiązanie

δ - krok o który będziemy powiększać początkowe liczby.
Obserwując otrzymywane wyniki, będziemy mogli dostować wielkość kroku, aby uzyskać zwiększenie o jeden bit na danym przedziale.

3.3 Wyniki

$\delta = 2^{-52}$ dla przedziału [1, 2]: Różnica po jednym kroku
0011111111110000000000000000000000000000000000001010110011000001 – 1.00000000000982
0011111111110000000000000000000000000000000000001010110011000010 – 1.0000000000098201
$\delta = \frac{2^{-52}}{2}$ dla przedziału $[\frac{1}{2}, 1]$: Tutaj musieliśmy podzielić krok przez 2 aby uzyskac taki sam rezultat jak wcześniej
0011111111100000000000000000000000000000000000001001001101001100 – 0.5000000000041864
0011111111100000000000000000000000000000000000001001001101001101 – 0.5000000000041865
$\delta = 2^{-52} * 2$ dla przedziału [2, 4]: Tutaj sytuacja odwrotna, długość kroku musiała zostać zwiększona
0100000000000000000000000000000000000000000000000111110111011101 – 2.000000000014309
0100000000000000000000000000000000000000000000000111110111011110 – 2.0000000000143094

3.4 Wnioski

W przedziale [1, 2] liczby występują co $\delta = 2^{-52}$.
W przedziale $[\frac{1}{2}, 1]$ 2 razy częściej, a w przedziale [2, 4] 2 razy rzadziej. Dla arytmetyki Float64 liczba bitów przeznaczonych na mantyse $t = 53$, dlatego gęstość występowania liczb w danym przedziale możemy przedstawić w następujący sposób: 2^{-t+d}, gdzie d to potęga liczby 2 przy górnej granicy przedziału.

4 Zadanie 4

4.1 Opis problemu

Znalezienie liczby w arytmetyce Float64 zgodnej ze standardem IEEE 754 liczbe zmiennopozycyjna x w przedziale $1 < x < 2$ taką, że $x * (1/x) \neq 1$.

4.2 Rozwiązanie

Algorithm 4 find

$delta \leftarrow 2^{-52}$
for k in $2^{-52} - 1$ **do**
  $x \leftarrow 1 + k * \delta$
  if $(x * \frac{1}{x}) \neq 1$ **then**
    return x
  end if
end for

4.3 Wyniki

Najmniejsza znaleziona wartość: 1.000000057228997

5 Zadanie 5

5.1 Opis problemu

Obliczenie 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]
używając pojedyńczej i podwójnej precyzji

5.2 Rozwiązanie

1. "w przód" t.j. $\sum_{i=1}^{n} x_i y_i$
2. "w tył" t.j. $\sum_{i=n}^{1} x_i y_i$
3. "liczby dodatnie od najwiekszego do najmniejszego a ujemne na odwrót"
4. "liczby ujemne od najwiekszego do najmniejszego a dodatnie na odwrót"

5.3 Wyniki

Źródło	Float32	Float64	Prawidłowy wynik
"1"	-0.4999443	1.0251881368296672e-10	-1.00657107000000e-11
"2"	-0.4543457	-1.5643308870494366e-10	-1.00657107000000e-11
"3"	-0.5	0.0	-1.00657107000000e-11
"4"	-0.5	0.0	-1.00657107000000e-11

5.3.1 Wnioski

Widzimy, że w zależności od obranego sposobu obliczania iloczynu skalarnego wyniki mniej lub bardziej odbiegają od rzeczywistości. Dla Float32 mamy bardzą dużą rozbieżność ze względu na zbyt niską precyzję arytmetyki. Na podstawie wyników możemy również stwierdzić, że sposób liczenia ma duży wpływ na dokładność naszego wyniku. Dodawanie "w przód" oraz "w tył" wygenerowało znacznie większy błąd względny niż w przypadku dwóch następnych metod. Widzimy, że prawidłowy wynik jest bardzo bliski zera co podpowiada nam dlaczego uzyskany wynik to 0.0 - nawet w przypadku Float64 precyzja jest zbyt niska.

6 Zadanie 6

6.1 Opis problemu

Policzenie wartości w arytmetyce Float64 następujących funkcji:

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

6.2 Wyniki

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

6.3 Wnioski

Widzimy, że funkcja $g(x)$ jest dokładniejsza, widać to przy największej i najmniejszych wartościach. Ujemny wynik zwrócony przez $f(x)$ nie powinien być możliwy do osiągnięcia, niestety ten wynik jak i ostatnie zerowe są spowodowane przez utratę precyzji przy odejmowaniy małych, bliskich siebie liczb. Zapisanie tej samej funkcji tak, aby uniknąć odejmowania skutecznie omija ten problem, który w rzeczywistości nie powinien mieć miejsca, gdyż są to dokładnie takie same funkcje. Pokazuje to jeden z problem w kalkulacjach w arytmetyce zmiennopozycyjnej.

7 Zadanie 7

7.1 Opis problemu

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

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

Korzystając z tego wzoru do obliczania w arytmetyce Float64 przybliżonej wartości pochodnej funckji $f(x) = sinx + cos3x$ w punkcie $x_0 = 1$ oraz obliczamy błędy $|f^{'}(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$ (n = 0, 1, 2, ..., 54)

7.2 Wyniki

h	aproksymacja	różnica od pochodnej
2^{-3}	0.6232412792975817	0.5062989976090435
2^{-7}	0.1484913953710958	0.03154911368255764
2^{-12}	0.11792723373901026	0.0009849520504721099
2^{-31}	0.11694216728210449	1.1440643366000813e-7
2^{-32}	0.11694192886352539	3.5282501276157063e-7

7.3 Wnioski

W wynikach możemy zaobserować ze od pewnego momentu zmniejszanie wartości h wcale nie poprawia przybliżenia wartości pochodnej. Prawdopodobnie znowu występuje tutaj problem odejmowania bliskich sobie liczb. Dodatkowo może pojawiać się utrata precyzji przy reprezentowaniu $x_0 + h$.