Agata Jasionowska 229726

Laboratorium – Lista 1

1. Zadanie 1

1.1. Macheps

1.1.1. Opis problemu

Napisanie w języku `Julia` programu wyznaczającego w sposób iteracyjny epsilony maszynowe (czyli najmniejsze takie liczby $macheps > 0$, że $1.0 + macheps > 1.0$) dla wszystkich dostępnych w tym języku typów zmiennopozycyjnych (`Float16`, `Float32` oraz `Float64`).

1.1.2. Opis rozwiązania

W celu znalezienia liczby *macheps* dla danego typu liczby zmiennopozycyjnej skorzystano z operacji przesunięcia bitowego w prawo zgodnie z podanymi niżej krokami:

1. Zdefiniowanie zmiennej $a = 1.0$ wybranego typu;
2. (W pętli) Dopóki $1.0 + a > 1.0$ przypisanie aktualnej wartości a w pomocniczej zmiennej b oraz wykonanie przesunięcia bitowego w prawo dla a.

Po zakończeniu wykonywania pętli w zmiennej a pozostanie wartość na tyle mała, że zostanie potraktowana jako zero maszynowe (czyli $1.0 + a = 1.0$). Zmienna b będzie przechowywała ostatnią przypisaną wartość większą niż 0.0, czyli poszukiwany *macheps*.

1.1.3. Wyniki

Uzyskano następujące wyniki dla kolejnych typów zmiennopozycyjnych:

typ	b	`eps(typ)`	`C`
`Float16`	9.7656e−4	9.7656e−4	9.77e−4
`Float32`	1.192 092 9e−7	1.192 092 9e−7	1.192 093e−7
`Float64`	2.220 446 049 250 313e−16	2.220 446 049 250 313e−16	2.220 446e−16

Tabela 1. Wyniki *macheps* wraz z prawidłowymi wartościami

1.1.4. Wnioski

Uzyskano rozwiązania identyczne ze zwracanymi przez funkcje `eps`, co dowodzi prawidłowości przyjętego sposobu rozwiązania problemu.
Precyzja arytmetyki(ϵ) ma wpływ na to, jak wiele cyfr znaczących liczby jest reprezentowanych dokładnie i zależy wyłącznie od liczby bitów przeznaczonych na reprezentację mantysy. W związku z tym istnieje jej bezpośredni związek z wartością epsilona maszynowego. Im mniejszy *macheps*, tym większa jest względna precyzja obliczeń.

1.2. ETA

1.2.1. Opis problemu

Napisanie w języku `Julia` programu wyznaczającego w iteracyjny sposób liczbę *eta* (taką, że $eta > 0.0$) dla dostępnych w nim typów zmiennopozycyjnych (`Float16`, `Float32` oraz `Float64`).

1.2.2. Opis rozwiązania

Algorytm wyznaczenia liczby *eta* jest zbliżony do algorytmu obliczającego wartości *macheps*:

Algorithm 1

$a \leftarrow 1.0$
while $a/2.0 > (0.0)$ **do**
  $a \leftarrow a/2.0$
end while

Po zakończeniu wykonywania pętli w zmiennej a pozostanie poszukiwana wartość *eta*.

1.2.3. Wyniki

Uzyskano następujące wyniki dla kolejnych typów zmiennopozycyjnych:

typ	a	`nextfloat(0.0)`	MIN_{SUB}
`Float16`	6.00e−8	6.00e−8	5.96e−8
`Float32`	1.0e−45	1.0e−45	1.4e−45
`Float64`	5.0e−324	5.0e−324	4.9e−324

Tabela 2. Wyniki *eta* wraz z prawidłowymi wartościami

1.2.4. Wnioski

Uzyskano rozwiązania identyczne ze zwracanymi przez funkcję `nextfloat()`, co dowodzi prawidłowości przyjętego sposobu rozwiązania problemu.
Liczba *eta* jest bardzo zbliżona do Min_{SUB}.

1.3. MAX

1.3.1. Opis problemu

Napisanie w języku `Julia` programu wyznaczającego w iteracyjny sposób liczbę MAX dla dostępnych w nim typów zmiennopozycyjnych (`Float16`, `Float32` oraz `Float64`).

1.3.2. Opis rozwiązania

W celu znalezienia wartości MAX dla danego typu liczby zmiennopozycyjnej skorzystano z operacji przesunięcia bitowego w prawo oraz z funkcji `isinf(`a`)` (zwracającej wartość *true*, jeżeli argument jest nieskończonością) zgodnie z podanymi niżej krokami:

1. Zdefiniowanie zmiennej $a = 1.0$ wybranego typu;
2. (W pętli) Dopóki $a * $ `FloatX`$(2.0)(gdzie X \in 16, 32, 64$ jest skończone, wykonanie przesunięcia bitowego w prawo dla a

Po zakończeniu wykonywania pętli w zmiennej a pozostanie wartość równa nieskończoności. Funkcja `prevfloat(`a`)` umożliwia odczytanie poprzedniej wartości zmiennopozycyjnej danego typu, czyli zadanej liczby MAX.

1.3.3. Wyniki

Uzyskano następujące wyniki dla kolejnych typów zmiennopozycyjnych:

typ	a	`realmax(typ)`	`C`
`Float16`	6.55e+4	6.55e+4	6.5504e+4
`Float32`	3.402 823 5e+38	3.402 823 5e+38	3.402 823 466 4e+38
`Float64`	1.797 693 134 862 315 7e+308	1.797 693 134 862 315 7e+308	1.797 69e+308

Tabela 3. Wyniki MAX wraz z prawidłowymi wartościami

1.3.4. Wnioski

Uzyskano rozwiązania identyczne ze zwracanymi przez funkcję `realmax()`, co dowodzi prawidłowości przyjętego sposobu rozwiązania problemu.
Wartości MAX dla kolejnych typów zmiennopozycyjnych są bardzo zbliżone do maksymalnych wartości deklarowanych w dokumentacji języka `C`.

2. Zadanie 2

2.1. Opis problemu

Napisanie w języku `Julia` programu, który eksperymentalnie sprawdzi słuszność stwierdzenia Kahana (epsilon maszynowy może zostać wyznaczony

w wyniku obliczenia $3(4/3-1)-1$ w danej arytmetyce zmiennopozycyjnej) dla wszystkich dostępnych typów zmiennopozycyjnych.

2.2. Opis rozwiązania

Obliczenie wartości wyrażenia z użyciem właściwego rzutowania typów zgodnie z poniższym wzorem:

`FloatX`(3) * ((`FloatX`(4) / `FloatX`(3)) - `FloatX` (1)) - `FloatX`(1), dla $X \in \{16, 32, 64\}$.

2.3. Wyniki

Uzyskano następujące wyniki dla kolejnych typów zmiennopozycyjnych:

typ	Kachan	*macheps*
`Float16`	−9.77e−4	9.77e−4
`Float32`	1.192 092 9e−7	1.192 092 9e−7
`Float64`	2.220 446 049 250 313e−16	2.220 446 049 250 313e−16

Tabela 4. Wyniki twierdzenia Kachana wraz z prawidłowymi wartościami

2.3.1. Wnioski

Powyższa tabela pokazuje, że prawidłowe rozwiązanie udało się uzyskać jedynie dla typu `Float32`. W dwóch pozostałych przypadkach wynik różnił się znakiem. Stwierdzenie Kachana byłoby słuszne, gdyby z wartości wyrażenia wziąć jego wartość bezwzględną.

3. Zadanie 3

3.1. Opis problemu

Napisanie w języku `Julia` programu, który eksperymentalnie sprawdzi, że w arytmetyce `Float64` liczby zmiennopozycyjne są równomiernie rozmieszczone w $[1, 2]$ z krokiem $\delta = 2^{-52}$. Równoznaczne jest to ze stwierdzeniem, iż każda liczby zmiennopozycyjna x z zakresu $[1, 2]$ może zostać przedstawiona jako $x = 1 + k * \delta$ w danej arytmetyce, dla $k = 1, 2, \ldots, 2^{52} - 1$ i $\delta = 2^{-52}$.

3.2. Opis rozwiązania

Eksperymentalne sprawdzenie rozmieszczenia liczb zgodnie z poniższym schematem:

1. Utworzenie takiej zmiennej δ, że $\delta = 2^k$, $k = 0, -1, -2 \ldots$ typu zmiennopozycyjnego;
2. Zdefiniowanie zmiennej a pierwszą wartością przedziału;
3. (W pętli) Zwiększanie a o wartość δ oraz wyświetlenie rezultatu wraz z jego zapisem bitowym (uzyskany przy pomocy funkcji `bits(`a`)`.

3.3. Wyniki

1. Przedział $[1, 2]$

a	zapis bitowy a
1.0	$001111111111000\ldots0001$
$1.0 + \delta$	$001111111111000\ldots0010$
$1.0 + 2 * \delta$	$001111111111000\ldots0011$
$\vdots$	$\vdots$
$2.0 - 3 * \delta$	$001111111111111\ldots1101$
$2.0 - 2\delta$	$001111111111111\ldots1110$
$2.0 - \delta$	$001111111111111\ldots1111$

Tabela 5. Rozmieszczenie liczb w zakresie $[1, 2]$ dla $\delta = 2^{-52}$

Zapis bitowy pokazuje, że dodawanie do liczby a wartości $\delta = 2^{-52}$ zwiększa ją o kolejny jeden bit.

2. Przedział $[0.5, 1]$

a	zapis bitowy a
0.5	$00111111111000\ldots00010$
$0.5 + \delta$	$00111111111000\ldots00100$
$0.5 + 2 * \delta$	$00111111111000\ldots00110$
$0.5 + 3 * \delta$	$00111111111000\ldots01000$
$\vdots$	$\vdots$
$1.0 - 3 * \delta$	$00111111111111\ldots11011$
$1.0 - 2\delta$	$00111111111111\ldots11101$
$1.0 - \delta$	$00111111111111\ldots11111$

Tabela 6. Rozmieszczenie liczb w zakresie $[0.5, 1]$ dla $\delta = 2^{-52}$

Analizę rozmieszczenia liczb rozpoczęto od zbadania zmian w zapisie bitowym dla $\delta = 2^{-52}$. Wyniki widoczne powyżej pokazują cykliczne zwiększanie się a o 2 bity, czyli wartości rozłożone są z dwukrotnie większym krokiem. Zatem rozmieszczenie dla tego przedziału to $\delta = \frac{1}{2} * 2^{-52} = 2^{-53}$.

3. Przedział $[2, 4]$
W ostatnim rozpatrywanym przedziale rezultaty dla $\delta = 2^{-52}$ uwidaczniają regularną zmianę bitów naprzemiennie co 1- oraz 3-krotne powiększenie a o wartość δ. Sugeruje to rozkład liczb z krokiem większym niż początkowo założony. Wartość zwiększono więc dwukrotnie δ i otrzymano wyniki:
Zatem rozkład liczb w $[2, 4]$ następuje z krokiem $\delta = 2^{-51}$.

a	zapis bitowy a
2.0	$01000\ldots0000$
$2.0+\delta$	$01000\ldots0000$
$2.0+2*\delta$	$01000\ldots0001$
$2.0+3*\delta$	$01000\ldots0010$
$2.0+4*\delta$	$01000\ldots0010$
$2.0+5*\delta$	$01000\ldots0010$

Tabela 7. Rozmieszczenie liczb w zakresie $[2,4]$ dla $\delta = 2^{-52}$

a	zapis bitowy a
2.0	$00111111111000\ldots00010$
$2.0+\delta$	$00111111111000\ldots00100$
$2.0+2*\delta$	$00111111111000\ldots00110$
$2.0+3*\delta$	$00111111111000\ldots01000$
$\vdots$	$\vdots$
$4.0-3*\delta$	$00111111111111\ldots11011$
$4.0-2\delta$	$00111111111111\ldots11101$
$4.0-\delta$	$00111111111111\ldots11111$

Tabela 8. Rozmieszczenie liczb w zakresie $[2,4]$ dla $\delta = 2^{-51}$

3.4. Wnioski

Analiza przypadku $[1,2]$ dowodzi równomiernego rozmieszczenia liczb w tym przedziale z krokiem $\delta = 2^{-52}$, czyli prawdziwy jest wzór: $x = 1 + k * \delta$. Zaobserwowano następujące rozmieszczenie w pozostałych przedziałach:
— $[0.5,1]$: $x = 1 + k * \delta, \delta = 2^{-53}$;
— $[2,4]$: $x = 1 + k * \delta, \delta = 2^{-51}$

4. Zadanie 4

4.1. Opis problemu

Napisanie w języku `Julia` programu znajdującego eksperymentalnie taką liczbę zmiennopozycyjną `Float64` $1 < x < 2$, że $x * (1/x) \neq 1$ (tj. $fl(xfl(1/x)) \neq 1$) oraz wyznaczenie najmniejszej takiej wartości.

4.2. Opis rozwiązania

Zastosowanie programu działającego zgodnie z poniższym pseudokodem:

Algorithm 2

$a \leftarrow$ `Float64`(1.0)
$b \leftarrow$ `Float64`(1.0)
while $a <$ `Float64`(2.0) **do**
  if $((b/a) * a \neq b)$ **then** *wypisz* a
  end if
  $a \leftarrow$ `nextfloat`(a)
end while

4.3. Wyniki

W wyniku kilkugodzinnej pracy program znalazł 2164117 rozwiązań (przy czym przy czym nie są to wszystkie) spełniających warunki zadania. Najmniejszym z nich jest liczba 1.000000057228997.

4.4. Wnioski

Eksperyment pozwala uzmysłowić, jak wiele istnieje liczb zmiennopozycyjnych pomiędzy każdą parą sąsiednich liczb całkowitych.

5. Zadanie 5

5.1. Opis problemu

Napisanie w języku `Julia` implementacji czterech algorytmów obliczających iloczyn skalarny dwóch zadanych wektorów: x =[2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957], y =[1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049] z użyciem typów `Float32` oraz `Float64`.

5.2. Opis rozwiązania

1. "w przód"
   $\sum_{i=1}^{n} x_i y_i$
2. "w tył"
   $\sum_{i=n}^{1} x_i y_i$

3. od największego do najmniejszego
   Obliczenie sumy tym algorytmem zostało zaimplementowane w następujący sposób (przykład kodu dla arytmetyki `Float32`):
4. od najmniejszego do największego
   Implementacja ostatniego z algorytmów jest analogiczna do kodu z poprzedniego podpunktu. Jedyna różnica polega tutaj na odpowiedniej kolejności sortowania tablicy z sumami częściowymi.

5.3. Wyniki

Poniższa tabela prezentuje uzyskane wyniki dla czterech algorytmów obliczających iloczyn skalarny:

podpunkt	`Float32`	`Float64`
a	$-0.499\,944\,3$	$1.025\,188\,136\,829\,667\,2\text{e}{-10}$
b	$-0.454\,345\,7$	$-1.564\,330\,887\,049\,436\,6\text{e}{-10}$
c	-0.5	0.0
d	-0.5	0.0

Tabela 9. Obliczanie iloczynu skalarnego wektorów

5.4. Wnioski

WYCIĄGNĄĆ WNIOSKI!!!

6. Zadanie 6

6.1. Opis problemu

Obliczenie w języku `Julia` 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 $x = 8^{-1}, 8^{-2}, \ldots$

6.2. Opis rozwiązania

Obliczanie wartości obu funkcji w pętli dla kolejnych argumentów.

6.3. Wyniki

Poniższa tabela prezentuje otrzymane rozwiązania:

x	$f(x)$	$g(x)$
8^{-1}	$7.7822185373186414e-3$	$7.7822185373187065e-3$
8^{-2}	$1.2206286282867573e-4$	$1.2206286282875901e-4$
8^{-3}	$1.9073468138230965e-6$	$1.907346813826566e-6$
$\vdots$	$\vdots$	$\vdots$
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	$4.336808689942018e-19$

Tabela 10. Wartości funkcji $f(x)$ oraz $g(x)$

6.4. Wnioski

Analiza uzyskanych rozwiązań pozwala zaobserwować, iż dla argumentu $1 < x < 8$ funkcje zwracają bardzo zbliżone wartości. Jednak dla $x > 8$ funkcja f zaczyna zwracać 0.0, zaś g podaje dokładny wynik. Pozwala to przypuszczać, że to właśnie jest bardziej wiarygodna funkcja.

7. Zadanie 7

7.1. Opis problemu

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

7.2. Opis rozwiązania

Utworzenie funkcji $f(x)$ oraz jej pochodnej $g(x)$, przy czym $g(x) = \cos x - 3 * \sin 3x$. Stworzenie pomocniczych funkcji: obliczającej przybliżoną pochodną oraz błąd pomiaru.

7.3. Wyniki

W wyniku pracy programu uzyskano następujące rezultaty:

n	$f`(x)$	*blad*
0	2.0179892252685967	$7.7822185373187065e-3$
1	1.8704413979316472	$1.2206286282875901e-4$
2	1.1077870952342974	$1.907346813826566e-6$
⋮	⋮	⋮
45	0.11328125	0.003661031688538152
46	0.109375	0.007567281688538152
47	0.109375	0.007567281688538152
48	0.09375	0.023192281688538152
49	0.125	0.008057718311461848
50	0.0	0.11694228168853815
51	0.0	0.11694228168853815
52	−0.5	0.6169422816885382
53	0.0	0.11694228168853815
54	0.0	0.11694228168853815

Tabela 11. Wartości funkcji $f(x)$ oraz $g(x)$

7.4. Wnioski

NAPISAĆ WNIOSKI!!! Jak wytłumaczyć, że od pewnego momentu zmniejszanie wartości h nie poprawia przybliżenia wartości pochodnej? Jak zachowują się wartości 1+h? Obliczone przybliżenia pochodnej porównać z dokładną wartością pochodnej, tj. zwróć uwagę na błędy $f`$ dla $h = 2 - n$ $(n = 0, 1, 2, \ldots, 54)$.