Obliczenia naukowe

Prowadzący: dr hab. Paweł Zieliński

wtorek TN 7^{30}

Sprawozdanie 1

Weronika Jasiak

236733

1. Zadanie 1
1.1. Epsilon maszynowy
1.1.1. Opis problemu
Napisanie programu w języku `Julia` wyznaczającego iteracyjnie epsilony maszynowe (najmniejsze liczby $macheps > 0$ takie, że $fl(1.0 + macheps) > 1.0$) dla wszystkich dostępnych typów zmiennopozycyjnych (`Float16`, `Float32`, `Float64`).
1.1.2. Rozwiązanie
Wyznaczając epsilon maszynowy w pierwszej kolejności należy zainicjować liczbę $eps = 1.0$, która jest zgodna ze wcześniej wybranym typem zmiennopozycyjnym, a następnie dzielić ją przez 2 tak długo jak nierówność $1 + \frac{eps}{2} > 1$ jest prawdziwa.

```
while 1 + eps / 2 > 1 do
    eps ← eta / 2
end
```

1.1.3. Wyniki
Dla poszczególnych typów zmiennopozycyjnych uzyskano następujące wyniki (Tabela 1).

typ	$macheps$	`eps(typ)`	`C`
`Float16`	0.000977	0.000977	-
`Float32`	1.1920929e-7	1.1920929e-7	1.19209289550781e-7
`Float64`	2.220446049250313e-16	2.220446049250313e-16	2.220446049250313e-16

Tabela 1. Wyniki m$acheps$, prawidłowe wartości funkcji wbudowanej w języku `Julia` oraz dane znajdujące się w pliku nagłówkowym float.h w języku `C`

1.1.4. Wnioski
Wyniki, które uzyskano podczas iteracyjnego wyznaczania epsilonu maszynowego są identyczne jak w przypadku wywołania wbudowanej funkcji `eps()` dostępnej w języku `Julia`, co dowodzi prawidłowości sposobu w jaki rozwiązano problem. Również wartości znajdujące się w pliku nagłówkowym `float.h` dowolnej instalacji języka `C` nie odbiegają znacząco od tych uzyskanych przy pomocy napisanego programu. Epsilon maszynowy zwany również $macheps$ jest niczym innym jak podwojoną precyzją arytmetyki, 2^{-t}, gdzie t jest liczbą cyfr mantysy f. Zatem im mniejsza jest wartość epsilonu maszynowego, tym większa jest precyzja obliczeń.
1.2. ETA
1.2.1. Opis problemu
Napisanie programu w języku `Julia` wyznaczającego iteracyjnie liczbę eta taką, że $eta > 0.0$ dla wszystkich typów zmiennopozycyjnych (`Float16`, `Float32`, `Float64`).
1.2.2. Rozwiązanie
Wyznaczając liczbę eta na samym początku należy przypisać jej wartość 1.0 zgodną z jednym z danych typów zmiennopozycyjnych, następnie w podobny sposób co w przypadku $macheps$ dzielić ja przez 2 tak długo jak nierówność $\frac{eta}{2} > 0$ jest prawdziwa.

```
while eta / 2 > 0 do
    eta ← eta / 2
end
```

1.2.3. Wyniki

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

typ	eta	`nextfloat(0.0)`
`Float16`	6.00e-8	6.00e-8
`Float32`	1.00e-45	1.00e-45
`Float64`	4.94e-324	4.94e-324

Tabela 2. Wyniki eta oraz prawidłowe wartości funkcji wbudowanej w języku `Julia`

1.2.4. Wnioski

Wyniki jakie zostały uzyskane podczas iteracyjnego wyznaczania liczby eta są takie same jak w przypadku wywołania funkcji `nextfloat()` dostępnej w języku `Julia`, co potwierdza poprawność sposobu w jaki rozwiązano problem.

1.3. MAX

1.3.1. Opis problemu

Napisanie programu w języku `Julia` wyznaczającego iteracyjnie liczbę (MAX) dla wszystkich typów zmiennopozycyjnych (`Float16`, `Float32`, `Float64`).

1.3.2. Rozwiązanie

Maksymalną wartość dla danego typu zmiennopozycyjnego można wyznaczyć przez zainicjowanie zmiennej $max = 2.0$ i mnożyć ją tak długo przez 2 dopóki $max \neq \infty$. W kolejnym kroku przypisujemy wartość $\frac{max}{2}$ do zmiennej $addition$, a następnie w kolejnych iteracjach pętli dodajemy tak długo max do $addition$ oraz dzielimy $addition$ przez 2 dopóki $max + addition \neq \infty$.

```
max ← Type(2.0)
while !isinf(max * 2) do
    max ← max * 2
end
addition ← Type(max / 2)
while !isinf(max + addition) do
    max ← max + addition
    addition ← addition / 2
end
```

1.3.3. Wyniki

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

typ	max	`realmax(typ)`	`C`
`Float16`	6.55e+4	6.55e+4	
`Float32`	3.4028235e38	3.4028235e38	3.402823466385288598e+38
`Float64`	1.7976931348623157e308	1.7976931348623157e308	1.797693134862315708+308

Tabela 3. Wyniki max, prawidłowe wartości funkcji wbudowanej w języku `Julia` oraz dane znajdujące się w pliku nagłówkowym float.h w języku `C`

1.3.4. Wnioski

Wyznaczając iteracyjnie liczbę (MAX) uzyskano identyczne wyniki jak te zwracane przez funkcję `realmax()` dostępną w języku `Julia.` Dane zawarte w pliku nagłówkowym `float.h` dowolnej instalacji języka `C` również mają przybliżone wartości do tych uzyskanych w sposób iteracyjny. Zatem sposób rozwiązania problemu jest prawidłowy.

2. Zadanie 2

2.1. Opis problemu

Napisanie programu w języku `Julia`, który sprawdzi eksperymentalnie słuszność stwierdzenia Kahana (epsilon maszynowy można otrzymać obliczając wyrażenie $3\left(\frac{4}{3}-1\right)-1$ w arytmetyce zmiennopozycyjnej) dla wszystkich typów zmiennopozycyjnych (`Float16`, `Float32`, `Float64`).

2.2. Rozwiązanie

Sprawdzając eksperymentalnie słuszność stwierdzenia Kahana wystarczy obliczyć wartość wyrażenia:

$$Type(3) * \left(\left(\frac{Type(4)}{Type(3)}\right) - Type(1)\right) - Type(1),$$

gdzie $Type$ to dostępne typy zmiennopozycyjne (`Float16`, `Float32`, `Float64`).

2.3. Wyniki

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

typ	Kahan	*macheps*
`Float16`	-0.000977	0.000977
`Float32`	$1.1920929\text{e}-7$	1.1920929e-7
`Float64`	-2.220446049250313e-16	2.220446049250313e-16

Tabela 4. Wyniki Khana oraz prawidłowe wartości funkcji wbudowanej w języku `Julia`

2.4. Wnioski

Uzyskane wyniki są rozbieżne w $66,(6)\%$, jedynie dla typu zmiennopozycyjnego `Float32` są one identyczne. Gdyby Kahan na samym końcu nałożył na powyższe wyrażenie wartość bezwzględną to jego stwierdzenie można by uznać za słuszne, a wyniki otrzymane w ten sposób byłyby zgodne z epsilonem maszynowym.

3. Zadanie 3

3.1. Opis problemu

Napisanie programu w języku `Julia`, który sprawdzi, że w arytmetyce `Float64` liczby zmiennopozycyjne są równomiernie rozmieszczone w $[1, 2]$ z krokiem $\delta = 2 - 52$. Oznacza to, że każda liczba zmiennopozycyjna x pomiędzy 1 i 2 może być przedstawiona w następujący sposób $x = 1 + k\delta$ w tej arytmetyce, dla $k = 1, 2, \ldots, 2^{52} - 1$ i $\delta = 2^{-52}$.

3.2. Rozwiązanie

Sprawdzając równomierność rozmieszczenia liczb w przedziałach można wykonać poniższe kroki:

1. Zainicjowanie zmiennych min i max, które są odpowiednio początkiem i końcem przedziału dla którego aktualnie sprawdzamy równomierne rozmieszczenie liczb oraz stałą $\delta = 2^{-52}$.
2. Zainicjowanie zmiennej $k = 1, 2, \dots, 2^{52} - 1$, według której będzie następować iteracja.
3. Zwiększanie wartości min o $k\delta$ dopóki nie osiągnie wartości max i wyświetlanie jej przy pomocy funkcji `bits()` pozwalającej zobaczyć zapis bitowy otrzymanych wyników.

3.3. Wyniki

Dla przedziału $[1, 2]$ (Tabela 5).

x	zapis bitowy x
$1.0 + 1 * \delta$	0011111111110000000000000000000000000000000000000000000000000001
$1.0 + 2 * \delta$	0011111111110000000000000000000000000000000000000000000000000010
$1.0 + 3 * \delta$	0011111111110000000000000000000000000000000000000000000000000011
$\vdots$	$\vdots$

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

Dla przedziału $[0.5, 1]$ (Tabela 6 i 7).

x	zapis bitowy x
$0.5 + 1 * \delta$	0011111111100000000000000000000000000000000000000000000000000010
$0.5 + 2 * \delta$	0011111111100000000000000000000000000000000000000000000000000100
$0.5 + 3 * \delta$	0011111111100000000000000000000000000000000000000000000000000110
$\vdots$	$\vdots$

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

Można zauważyć, że powyższe wyniki (Tabela 6) zwiększają się o dwa bity, czyli liczby znajdujące się w tym przedziale rozłożone są z dwukrotnie większym krokiem. Zatem, aby rozmieszczenie tego przedziału było poprawne należy zmienić krok $\delta = 2^{-52}$ na $\delta = 2^{-53}$ (Tabela 7).

x	zapis bitowy x
$0.5 + 1 * \delta$	0011111111100000000000000000000000000000000000000000000000000001
$0.5 + 2 * \delta$	0011111111100000000000000000000000000000000000000000000000000010
$0.5 + 3 * \delta$	0011111111100000000000000000000000000000000000000000000000000011
$\vdots$	$\vdots$

Tabela 7. Rozmieszczenie liczb w zakresie $[0.5, 1]$ dla $\delta = 2^{-53}$

Dla przedziału $[2.0, 4.0]$ (Tabela 8 i 9).

x	zapis bitowy x
$2.0 + 1 * \delta$	0100000000000000000000000000000000000000000000000000000000000000
$2.0 + 2 * \delta$	0100000000000000000000000000000000000000000000000000000000000000
$2.0 + 3 * \delta$	0100000000000000000000000000000000000000000000000000000000000010
$\vdots$	$\vdots$

Tabela 8. Rozmieszczenie liczb w zakresie $[2.0, 4.0]$ dla $\delta = 2^{-52}$

Można zauważyć, że wyniki z tego przedziału (Tabela 8) różnią się od tych z pierwszego (Tabela 5), ponieważ mamy tutaj dwukrotnie większy zakres, czyli gęstość liczb jest dwa razy większa. Zatem, aby rozmieszczenie tego przedziału było poprawne należy zmienić krok $\delta = 2^{-52}$ na $\delta = 2^{-51}$ (Tabela 9).

x	zapis bitowy x
$2.0 + 1 * \delta$	0100000000000000000000000000000000000000000000000000000000000001
$2.0 + 2 * \delta$	0100000000000000000000000000000000000000000000000000000000000010
$2.0 + 3 * \delta$	0100000000000000000000000000000000000000000000000000000000000011
$\vdots$	$\vdots$

Tabela 9. Rozmieszczenie liczb w zakresie $[2.0, 4.0]$ dla $\delta = 2^{-51}$

3.4. Wnioski

Liczby w pierwszym przedziale są równomiernie rozmieszczone z krokiem $\delta = 2^{-52}$. W kolejnych przedziałach należało zmienić krok δ, aby uzyskać równomierne rozmieszczenie korzystając z tego samego wzoru $x = 1 + k\delta$ dla wszystkich przypadków. Zatem czym większa odległość od 0.0, tym większa odległość pomiędzy kolejnymi liczbami, co prowadzi do zwiększania się delty powodując mniejszą precyzję w arytmetyce `Float64`

4. Zadanie 4

4.1. Opis problemu

Napisanie programu w języku `Julia`, który wyznaczy eksperymentalnie taką liczbę zmiennopozycyjną `Float64` x znajdującą się w przedziale $1 < x < 2$, że $x * \left(\frac{1}{x}\right) \neq 1$ (tj. $fl(xfl\left(\frac{1}{x}\right) \neq 1)$ oraz wyznaczenie najmniejszej takiej wartości.

4.2. Rozwiązanie

Wyznaczając liczbę x należy zainicjować zmienną x i y, które przyjmują odpowiednio wartości początku i końca przedziału, a następnie dopóki $x < y$ należy przypisywać x kolejne wartości z przedziału, gdy $x * \left(\frac{1}{x}\right) \neq 1$ zostaje przerwana praca programu i następuje wypisanie wartości x.

```
while x < y do
    if x * (1 / x) != 1 do
        break
    end
    x ← nextfloat(x)
end
```

4.3. Wyniki

Dla poszczególnych przedziałów uzyskano następujące wyniki (Tabela 10).

$1 < x < 2$	1.000000057228997
$-\infty < x < \infty$	-1.7976931348623157e308

Tabela 10. Wyniki spełniające założenie $x * \left(\frac{1}{x}\right) \neq 1$ dla podanego przedziału

4.4. Wnioski

Znajdujące się powyżej wyniki (Tabela 10) są najmniejszymi wartościami spełniającymi warunek $x * \left(\frac{1}{x}\right) \neq 1$ dla podanych przedziałów, jednak takich liczb jest znacznie więcej. Zaokrąglając liczbę traci się precyzję.

5. Zadanie 5

5.1. Opis problemu

Implementacja czterech algorytmów w języku `Julia`, które obliczają iloczyn skalarny dwóch zadanych wektorów x i y przy użyciu typów zmiennopozycyjnych `Float32` oraz `Float64`.

5.2. Rozwiązanie

Obliczając iloczyn skalarny dwóch zadanych wektorów na samym początku należy zainicjalizować wartość zmiennej $n = 5$, według której będą następować iteracje pętli, a w następnym kroku podać wartości tablic dla wektorów x i y oraz zainicjalizować zmienną S przechowującą sumę ich iloczynów.

5.2.1. „W przód”

```
for i ← 1 to n do
    S ← S + Type(x[i] * y[i])
end
```

5.2.2. „W tył”

```
i ←  n
while i != 0 do
    S ← S + Type(x[i] * y[i])
    i ← i - 1
end
```

5.2.3. Od największego

```
product ← Type[]
for i ←  1 to n do
    push!(product, (x[i] * y[i]))
end
sort!(product, rev ← true)
positiveS ← Type(0.0)
for i ← 1 to n do
    if product[i] > 0 do
        positiveS ← positiveS + product[i]
    end
end
sort(product)
negativeS ← Type(0.0)
for i ← 1 to n do
    if product[i] < 0 do
        negativeS ← negativeS + product[i]
    end
end
S ← positiveS + negativeS
```

5.2.4. Od najmniejszego

Algorytm działa w ten sam sposób jak algorytm z punktu 5.2.3., jedyne zmiany jakie w nim zachodzą to zmiany sposobu sortowania.

5.3. Wyniki

Dla poszczególnych typów zmiennopozycyjnych uzyskano następujące wyniki (Tabela 11).

	`Float32`	`Float64`
1	-0.4999443	1.0251881368296672e-10
2	-0.4543457	-1.5643308870494366e-10
3	-0.5	0.0
4	-0.5	0.0

Tabela 11. Produkt iloczynu skalarnego wektorów na cztery różne sposoby

5.4. Wnioski

Otrzymane wyniki odbiegają od rzeczywistej wartości $-1.00657107000000010_{10} - 11$, jednak najbliższym wynikiem do podanej w poleceniu wartości jest algorytm „W tył" dla typu zmiennopozycyjnego `Float64`. Dodając posortowane liczby do siebie można zauważyć, że następuje strata precyzji w wyniku pochłonięcia mniejszej liczby przez większą.

6. Zadanie 6

6.1. Opis problemu

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

6.2. Rozwiązanie

Obliczenie wartości następuje w pętli, gdzie zmieniane są jedynie wartości potęgi argumentu x i następuje wypisanie wyników.

6.3. Wyniki

Dla poszczególnych funkcji uzyskano następujące wyniki (Tabela 12).

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
$\vdots$	$\vdots$	$\vdots$
8^{-8}	1.7763568394002505e-15	1.7763568394002489e-15
8^{-9}	0.0	2.7755575615628914e-17
8^{-10}	0.0	4.336808689942018e-19
$\vdots$	$\vdots$	$\vdots$

Tabela 12. Wartości funkcji $f(x)$ i $g(x)$ w kolejnych iteracjach

6.4. Wnioski

Zarówno funkcja $f(x)$ jak i $g(x)$ przyjmują podobne wartości dla pierwszych ośmiu argumentów, jednak gdy następuje dziewiąta iteracja pętli funkcja $f(x)$ zaczyna przyjmować wartość 0.0, podczas gdy funkcja $g(x)$ zwraca wartość. Można więc zauważyć, że funkcja $g(x)$ jest o wiele dokładniejsza, ponieważ przyjmuje ona dopiero wartość 0.0 przy wykładniku -179 pomimo, że $f(x) = g(x)$.

7. Zadanie 7

7.1. Opis problemu

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

7.2. Rozwiązanie

Na samym początku należy zainicjalizować funkcję $f(x)$ oraz jej pochodną $g(x)$, następnie w pętli zostaje obliczona przybliżona wartość pochodnej w punkcie $x_0 = 1$ oraz błąd dla $h = 2^{-n}$ $(n = 0, 1, 2, \ldots, 54)$.

```
x ← Float64(1.0)
for i ← 0 to n do
    h ← Float64(2.0)^(-i)
    approximateDerivative ← ((f(x + h) - f(x)) / h)
    approximateError ← abs(g(x) - approximateDerivative)
end
```

7.3. Wyniki

Dla poszczególnych iteracji uzyskano następujące wyniki (Tabela 12).

i	$f'(x)$	błąd
0	2.0179892252685967	1.9010469435800585
1	1.8704413979316472	1.753499116243109
2	1.1077870952342974	0.9908448135457593
3	0.6232412792975817	0.5062989976090435
⋮	⋮	⋮
51	0.0	0.11694228168853815
52	-0.5	0.6169422816885382
53	0.0	0.11694228168853815
54	0.0	0.11694228168853815

Tabela 13. Wartości funkcji $f(x)$ oraz błąd w kolejnych iteracjach

7.4. Wnioski

Wyniki otrzymane podczas kolejnych iteracji pętli nasuwają na myśl wniosek, że wraz ze zmniejszaniem się liczby h wyrażenie $h + 1$ zaczyna przyjmować wartość 1. Zatem gdy liczba h jest już tak mała, że nie jest w stanie wpłynąć na wykonywanie działanie, to wartości wykonywanych obliczeń ulegają zburzeniu.