Sprawozdanie z Laboratorium Metod Numerycznych

Marcel Musiałek
279704

27 października 2025

Spis treści

1 Zadanie 1: Rozpoznanie arytmetyki

1.1 Część 1: Wyznaczanie epsilona maszynowego (*macheps*)

1.1.1 Opis problemu

Celem było iteracyjne wyznaczenie epsilona maszynowego (*macheps*), czyli najmniejszej liczby $x > 0$ takiej, że $1.0 + x \neq 1.0$. Zadanie wymagało porównania uzyskanej wartości z wbudowaną funkcją Julii `eps(T)` oraz standardowymi stałymi z pliku `float.h` języka C dla typów `Float16`, `Float32` i `Float64`.

1.1.2 Opis rozwiązania

Zaimplementowano funkcję `find_macheps(T)`. Funkcja inicjuje `macheps` jako 1.0 danego typu `T`. Następnie, w pętli `while`, wartość `macheps` jest iteracyjnie dzielona przez 2.0. Pętla kontynuuje działanie tak długo, jak długo warunek $1.0 + (macheps/2.0) > 1.0$ jest spełniony w arytmetyce typu `T`. Zwracana jest ostatnia wartość `macheps`, dla której warunek ten nie był już prawdziwy.

1.1.3 Wyniki i interpretacja

Poniżej przedstawiono wyniki uzyskane z implementacji iteracyjnej oraz z funkcji wbudowanej.

```
--- Część 1: Wyznaczanie Epsilona Maszynowego (macheps) ---
--- Typ: Float16 ---
Iteracyjnie:           9.7656250000e-04
Wbudowane (eps(T)):    9.7656250000e-04
--- Typ: Float32 ---
Iteracyjnie:           1.1920928955e-07
Wbudowane (eps(T)):    1.1920928955e-07
--- Typ: Float64 ---
Iteracyjnie:           2.2204460493e-16
Wbudowane (eps(T)):    2.2204460493e-16
```

Interpretacja i porównanie z `float.h` (C): Wyniki obliczeń iteracyjnych są identyczne z funkcjami wbudowanymi Julii. Poniżej jawne zestawienie ze standardowymi stałymi C:

- **Float32 (Single):**
  - Nasz wynik: `1.1920928955e-07`
  - Stała C `FLT_EPSILON`: `1.1920928955e-07` (2^{-23})
  - *Zgodność: Tak*
- **Float64 (Double):**
  - Nasz wynik: `2.2204460493e-16`
  - Stała C `DBL_EPSILON`: `2.2204460493e-16` (2^{-52})
  - *Zgodność: Tak*

1.1.4 Wnioski

Obliczenia iteracyjne dały wyniki w pełni zgodne z wbudowanymi funkcjami Julii oraz standardowymi wartościami `FLT_EPSILON` i `DBL_EPSILON` ze **standardu IEEE 754**. Wartość ta definiuje *jednostkowy błąd zaokrąglenia* ($u = macheps/2$), który jest kluczowym parametrem w analizie błędów **obliczeń maszynowych** i pokazuje granicę precyzji względnej operacji.

1.2 Część 2: Wyznaczanie najmniejszej liczby dodatniej (η / MIN_{sub})

1.2.1 Opis problemu

Zadanie polegało na iteracyjnym wyznaczeniu najmniejszej dodatniej liczby maszynowej η (najmniejszej liczby subnormalnej, MIN_{sub}) poprzez dzielenie 1.0 przez 2.0 aż do osiągnięcia 0.0. Wynik należało porównać z funkcją `nextfloat(T(0.0))`.

1.2.2 Opis rozwiązania

Zaimplementowano funkcję `find_eta(T)`. Funkcja inicjuje η jako 1.0 danego typu `T`. W pętli `while`, sprawdzany jest warunek `(eta / 2.0) > 0.0`. Jeśli jest prawdziwy, η przyjmuje wartość `eta / 2.0`. Pętla zatrzymuje się, gdy następny krok dałby zero. Zwracana jest ostatnia wartość η, która była większa od zera.

1.2.3 Wyniki i interpretacja

Otrzymane wyniki z algorytmu iteracyjnego oraz funkcji wbudowanej:

```
--- Część 2: Wyznaczanie Najmniejszej Liczby Dodatniej (eta / MIN_sub) ---
--- Typ: Float16 ---
Iteracyjnie:                 5.9604644775e-08
Wbudowane (nextfloat(0.0)):    5.9604644775e-08
--- Typ: Float32 ---
Iteracyjnie:                 1.4012984643e-45
Wbudowane (nextfloat(0.0)):    1.4012984643e-45
--- Typ: Float64 ---
Iteracyjnie:                 4.9406564584e-324
Wbudowane (nextfloat(0.0)):    4.9406564584e-324
```

Interpretacja i porównanie z `float.h` (C): Wyniki iteracyjne są identyczne z wartościami `nextfloat(T(0.0))`. Standardowy plik `float.h` (np. C99) nie definiuje stałych dla najmniejszych liczb subnormalnych; definiuje jedynie `FLT_MIN` i `DBL_MIN`, które odpowiadają liczbom znormalizowanym (patrz Część 3).

1.2.4 Wnioski

Eksperyment potwierdził istnienie **liczb subnormalnych (zdenormalizowanych)**, kluczowego elementu **standardu IEEE 754**. Wyznaczona η (MIN_{sub}) reprezentuje mechanizm "łagodnego niedomiaru"(gradual underflow). W **obliczeniach maszynowych** pozwala to uniknąć gwałtownego "przeskoku"do zera, co jest krytyczne dla stabilności algorytmów operujących na bardzo małych wartościach.

1.3 Część 3: Analiza `floatmin` vs MIN_{nor} (i `float.h`)

1.3.1 Opis problemu

Celem było sprawdzenie, co zwraca funkcja `floatmin(T)` i jaki ma związek z MIN_{nor} (najmniejszą liczbą znormalizowaną) oraz ze stałymi `FLT_MIN` i `DBL_MIN` z `float.h`.

1.3.2 Opis rozwiązania

W tej części nie implementowano nowej funkcji iteracyjnej. Skrypt wywołał wbudowaną funkcję Julii `floatmin(T)` oraz (ponownie) funkcję `find_eta(T)` z Części 2, aby bezpośrednio zestawić ich wartości na wyjściu terminala w celu porównania.

1.3.3 Wyniki i interpretacja

Wyniki porównania funkcji `floatmin(T)` z obliczoną wcześniej η (MIN_{sub}):

```
--- Część 3: Badanie floatmin(T) vs eta ---
floatmin(Float16):     6.1035156250e-05 (vs eta: 5.9604644775e-08)
floatmin(Float32):     1.1754943508e-38 (vs eta: 1.4012984643e-45)
floatmin(Float64):     2.2250738585e-308 (vs eta: 4.9406564584e-324)
```

Interpretacja i porównanie z `float.h` (C): Jak widać na wydruku, wartości `floatmin(T)` są znacznie większe niż η (MIN_{sub}). Porównanie ze stałymi C:

- **Float32 (Single):**
  - Nasz wynik (`floatmin`): `1.1754943508e-38`
  - Stała C `FLT_MIN`: `1.1754943508e-38`
  - *Zgodność: Tak*
- **Float64 (Double):**
  - Nasz wynik (`floatmin`): `2.2250738585e-308`
  - Stała C `DBL_MIN`: `2.2250738585e-308`
  - *Zgodność: Tak*

1.3.4 Wnioski

Eksperyment jasno pokazuje fundamentalne rozróżnienie w **standardzie IEEE 754**:

1. η (MIN_{sub}): Najmniejsza liczba subnormalna (obszar *gradual underflow*).
2. `floatmin(T)` (MIN_{nor}): Najmniejsza liczba **znormalizowana**.

Jak widać z jawnego porównania, to właśnie MIN_{nor} (a nie MIN_{sub}) jest zdefiniowane w standardzie C (`FLT_MIN`, `DBL_MIN`). W **obliczeniach maszynowych** MIN_{nor} wyznacza początek zakresu, w którym liczby mają pełną precyzję względną.

1.4 Część 4: Wyznaczanie największej liczby skończonej (MAX)

1.4.1 Opis problemu

Zadanie polegało na iteracyjnym wyznaczeniu największej skończonej liczby maszynowej (MAX). Algorytm miał znaleźć największą potęgę dwójki $P = 2^{E_{max}}$ (mnożąc do `Inf`), a następnie obliczyć $MAX = P \times (2.0 - macheps)$. Wynik należało porównać z `floatmax(T)` oraz stałymi z `float.h`.

1.4.2 Opis rozwiązania

Zaimplementowano funkcję `find_max(T)`. Najpierw wywołuje ona funkcję `find_macheps(T)` z Części 1. Następnie, w pętli `while`, inicjalizuje `max_power_of_2` jako 1.0 i mnoży tę wartość przez 2.0, dopóki sprawdzenie `!isinf(max_power_of_2 * 2.0)` jest prawdziwe. Ostatnia skończona potęga dwójki (P) jest następnie używana we wzorze $MAX = P \times (2.0 - macheps)$ do obliczenia wyniku.

1.4.3 Wyniki i interpretacja

Wyniki iteracyjnego obliczania MAX w porównaniu z funkcją wbudowaną:

```
--- Część 4: Wyznaczanie Największej Liczby Skończonej (MAX) ---
--- Typ: Float16 ---
Iteracyjnie:           6.5504000000e+04
Wbudowane (floatmax):    6.5504000000e+04
--- Typ: Float32 ---
Iteracyjnie:           3.4028234664e+38
Wbudowane (floatmax):    3.4028234664e+38
--- Typ: Float64 ---
Iteracyjnie:           1.7976931349e+308
Wbudowane (floatmax):    1.7976931349e+308
```

Interpretacja i porównanie z `float.h` (C): Wyniki iteracyjne są identyczne z wartościami `floatmax(T)`. Porównanie ze stałymi C:

- **Float32 (Single):**
  - Nasz wynik: `3.4028234664e+38`
  - Stała C `FLT_MAX`: `3.4028234664e+38`
  - *Zgodność: Tak*
- **Float64 (Double):**
  - Nasz wynik: `1.7976931349e+308`
  - Stała C `DBL_MAX`: `1.7976931349e+308`
  - *Zgodność: Tak*

1.4.4 Wnioski

Wyznaczona wartość MAX reprezentuje górną granicę zakresu liczb znormalizowanych. Jawne porównanie potwierdza pełną zgodność ze **standardem IEEE 754** oraz stałymi C (`FLT_MAX`, `DBL_MAX`). W **obliczeniach maszynowych**, przekroczenie tej wartości skutkuje "nadmiarem"(overflow) i jest sygnalizowane wartością `Inf`. Zrozumienie tego limitu jest niezbędne przy projektowaniu stabilnych numerycznie algorytmów.

2 Zadanie 2: Obliczanie macheps metodą Kahana

2.0.1 Opis problemu

Celem zadania była eksperymentalna weryfikacja formuły Williama Kahana służącej do wyznaczania epsilona maszynowego: $3 \times (4/3 - 1) - 1$. Testy przeprowadzono dla typów `Float16`, `Float32` i `Float64`, porównując uzyskany wynik z wbudowaną funkcją `eps(T)`.

2.0.2 Opis rozwiązania

Zaimplementowano funkcję `kahan_macheps(T)`. Kluczowe było, aby wszystkie stałe (1.0, 3.0, 4.0) oraz wszystkie operacje pośrednie były wykonywane ściśle w arytmetyce badanego typu `T`. Funkcja krok po kroku obliczała $fl(4/3)$, następnie $fl(fl(4/3) - 1)$, $fl(3 \times \dots)$ i na końcu odejmowała 1.0, co pozwoliło na precyzyjne uchwycenie błędów zaokrągleń specyficznych dla każdej precyzji.

2.0.3 Wyniki i interpretacja

Poniżej przedstawiono wyniki uzyskane z implementacji formuły Kahana oraz wartości z funkcji wbudowanej.

```
--- Zadanie 2: Sprawdzanie formuły Kahana dla macheps ---

--- Typ: Float16 ---
Wynik z formuły Kahana:       -9.7656250000e-04
Wbudowane (eps(T)):          9.7656250000e-04
Czy równe eps(T)?           false
Czy równe -eps(T)?          true

--- Typ: Float32 ---
Wynik z formuły Kahana:       1.1920928955e-07
Wbudowane (eps(T)):          1.1920928955e-07
Czy równe eps(T)?           true
Czy równe -eps(T)?          false

--- Typ: Float64 ---
Wynik z formuły Kahana:       -2.2204460493e-16
Wbudowane (eps(T)):          2.2204460493e-16
Czy równe eps(T)?           false
Czy równe -eps(T)?          true
```

Interpretacja: Eksperyment dał zaskakujący i bardzo pouczający wynik. Wartość bezwzględna formuły Kahana jest zawsze równa `eps(T)`, jednak znak wyniku zależy od typu zmiennoprzecinkowego.

- **Float16:** Wynik jest równy `-eps(T)`.
- **Float32:** Wynik jest równy `eps(T)`.
- **Float64:** Wynik jest równy `-eps(T)`.

2.0.4 Wnioski

Stwierdzenie Kahana jest słuszne co do *wartości bezwzględnej* – formuła ta doskonale izoluje błąd zaokrąglenia o wielkości *macheps*. Różnica w znaku jest fascynującą ilustracją subtelności reguł zaokrąglania w **standardzie IEEE 754**.

Kluczowa jest pierwsza operacja: $fl(4/3)$. Liczba 4/3 ma nieskończone rozwinięcie binarne: $1.01010101\ldots_2$. Standard **IEEE 754** stosuje zaokrąglanie do najbliższej wartości ("round-to-nearest").

1. **Przypadek `Float16` ($p = 11$ bitów) i `Float64` ($p = 53$ bity):** Dla obu tych precyzji, odcięta część rozwinięcia binarnego jest mniejsza niż połowa jednostki na ostatnim miejscu (ULP). Liczba 4/3 jest zaokrąglana **w dół**. Analiza błędu pokazuje, że $fl(4/3) = 4/3 - \epsilon_1$ (błąd początkowy jest ujemny), co po kolejnych operacjach i końcowej anulacji daje wynik $-macheps$.

2. **Przypadek `Float32` ($p = 24$ bity):** Dla tej konkretnej precyzji, odcięta część rozwinięcia jest większa niż połowa ULP. Liczba 4/3 jest zaokrąglana **w górę**. Analiza błędu pokazuje, że $fl(4/3) = 4/3 + \epsilon_1$ (błąd początkowy jest dodatni), co po propagacji błędu i końcowej anulacji daje wynik $+macheps$.

Eksperyment ten pokazuje, że kierunek zaokrąglenia w **obliczeniach maszynowych** zależy od konkretnej liczby bitów precyzji, co może prowadzić do pozornie sprzecznych (ale poprawnych) wyników dla różnych typów zmiennoprzecinkowych.

---

3 Zadanie 3: Rozmieszczenie liczb zmiennoprzecinkowych

3.1 Część 3.1: Badanie przedziału [1, 2]

3.1.1 Opis problemu

Zadanie polegało na eksperymentalnym sprawdzeniu, czy w arytmetyce `Float64` (standard IEEE 754) liczby zmiennoprzecinkowe w przedziale $[1, 2]$ są równomiernie rozmieszczone. Mieliśmy zweryfikować, czy krok (odstęp) między kolejnymi liczbami jest stały i wynosi $\delta = 2^{-52}$. Weryfikacja opierała się na obliczeniu odstępu między 1.0 a następną liczbą maszynową oraz na analizie reprezentacji binarnej (używając `bitstring`).

3.1.2 Opis rozwiązania

Zdefiniowano teoretyczną wartość $\delta = 2^{-52}$ i porównano ją z `eps(1.0)`. Następnie zbadano liczbę $x_1 = 1.0$ i jej reprezentację binarną. Kolejna liczba, x_2, została wyznaczona na dwa sposoby: (1) przez dodanie delty $(1.0 + \delta)$ oraz (2) przy użyciu funkcji `nextfloat(1.0)`. Porównano wyniki i ich reprezentacje binarne. Procedurę powtórzono dla x_3 (krok $k = 2$), porównując $1.0 + 2\delta$ z `nextfloat(nextfloat(1.0))`.

3.1.3 Wyniki i interpretacja

Poniżej przedstawiono kluczowe wyniki uzyskane z implementacji.

```
--- Część 3.1: Badanie przedziału [1.0, 2.0] ---
Teoretyczna delta (2^-52):     2.220446049250313e-16
Wartość eps(1.0):              2.220446049250313e-16

Liczba x1 = 1.0 (k=0)
Bitstring x1:   0 01111111111 0000000000000000000...

Liczba x2 (następna po 1.0, k=1)
Bitstring x2 (z nextfloat):   0 01111111111 0000000000000000000...0001
Bitstring x2 (z 1.0 + delta):   0 01111111111 0000000000000000000...0001
Wartość x2 (z nextfloat):       1.0000000000000002e+00
Czy (1.0 + 2^-52) == nextfloat(1.0)?   true

Liczba x3 (k=2)
Bitstring x3 (z nextfloat):   0 01111111111 0000000000000000000...0010
Bitstring x3 (z 1.0 + 2*delta):   0 01111111111 0000000000000000000...0010
Czy (1.0 + 2*2^-52) == nextfloat(nextfloat(1.0))?   true
```

Interpretacja: Wyniki w pełni potwierdzają hipotezę.

1. **Zgodność δ:** Teoretyczna wartość $\delta = 2^{-52}$ jest identyczna z wartością `eps(1.0)`.
2. **Analiza `bitstring`:** $x_1 = 1.0$ ma wykładnik 2^0 (zapisany jako `01111111111`) i zerową mantysę. Obie metody wyznaczenia x_2 (dla $k = 1$) dały identyczny wynik, w którym zmienił się tylko ostatni bit mantysy (`...000` → `...001`). Dla $k = 2$, mantysa poprawnie zmieniła się na `...010`.

3. **Wniosek częściowy:** Wszystkie liczby w przedziale $[1, 2)$ mają ten sam wykładnik $E = 0$. Wartość liczby to $1.F \times 2^0$. Zmiana ostatniego bitu mantysy (o wadze 2^{-52}) przesuwa nas do kolejnej liczby maszynowej.

3.1.4 Wnioski

Eksperyment potwierdził, że w **standardzie IEEE 754** dla `Float64`, liczby w przedziale $[1, 2]$ są rozmieszczone **równomiernie (liniowo)**.

- Odstęp między nimi (ULP - Unit in the Last Place) jest stały i wynosi $\delta = 2^{-52}$, co jest równe `eps(1.0)` (czyli *macheps*).
- Każda liczba x w tym przedziale może być zapisana jako $x = 1.0 + k \cdot 2^{-52}$, gdzie $k \in \{0, 1, \ldots, 2^{52}\}$.
- W **obliczeniach maszynowych** oznacza to, że błąd bezwzględny w tym przedziale jest stały.

3.2 Część 3.2: Badanie przedziałów [0.5, 1] oraz [2, 4]

3.2.1 Opis problemu

Należało zbadać, jak rozmieszczone są liczby `Float64` w sąsiednich przedziałach potęg dwójki: $[0.5, 1]$ oraz $[2, 4]$. Celem było znalezienie kroku δ dla każdego z tych przedziałów i określenie ogólnej formy reprezentacji liczb.

3.2.2 Opis rozwiązania

Dla przedziału $[2, 4]$ zbadano liczbę $y_1 = 2.0$ oraz $y_2 =$ `nextfloat(2.0)`. Obliczono krok $\delta_y = y_2 - y_1$ i porównano go z `eps(2.0)` oraz 2^{-51}. Zanalizowano reprezentacje binarne y_1 i y_2. Analogiczną procedurę zastosowano dla przedziału $[0.5, 1]$, badając $z_1 = 0.5$, $z_2 =$ `nextfloat(0.5)` i obliczając $\delta_z = z_2 - z_1$.

3.2.3 Wyniki i interpretacja

Poniżej przedstawiono wyniki dla obu przedziałów.

```
--- Część 3.2: Badanie przedziałów [2.0, 4.0] i [0.5, 1.0] ---

--- Przedział [2.0, 4.0] ---
Liczba y1 = 2.0
Bitstring y1:   0 10000000000 0000000000000000000...
Liczba y2 (następna po 2.0)
Bitstring y2:   0 10000000000 0000000000000000000...0001

Obliczony krok delta_y = y2-y1:       4.440892098500626e-16
Wartość eps(2.0):                4.440892098500626e-16
Czy delta_y == 2^-51?          true

--- Przedział [0.5, 1.0] ---
Liczba z1 = 0.5
Bitstring z1:   0 01111111110 0000000000000000000...
Liczba z2 (następna po 0.5)
```

```
Bitstring z2:   0 01111111110 0000000000000000000...0001

Obliczony krok delta_z = z2-z1:       1.11022302462515654e-16
Wartość eps(0.5):                1.11022302462515654e-16
Czy delta_z == 2^-53?         true
```

Interpretacja:

- **Przedział [2.0, 4.0]:** $y_1 = 2.0$ ma wykładnik 2^1 (zapisany jako `100...000`). Kolejna liczba y_2 ma ten sam wykładnik i mantysę `...001`. Obliczony krok δ_y jest identyczny z `eps(2.0)` i 2^{-51}.
- **Przedział [0.5, 1.0]:** $z_1 = 0.5$ ma wykładnik 2^{-1} (zapisany jako `011...110`). Kolejna liczba z_2 ma ten sam wykładnik i mantysę `...001`. Obliczony krok δ_z jest identyczny z `eps(0.5)` i 2^{-53}.

3.2.4 Wnioski

Eksperyment pokazał, że liczby zmiennoprzecinkowe nie są rozmieszczone równomiernie w całym zakresie liczb rzeczywistych, ale są równomiernie (liniowo) rozmieszczone wewnątrz przedziałów $[2^E, 2^{E+1})$.

- W przedziale $[2, 4]$ ($E = 1$), krok $\delta_y = 2^{-51}$, czyli $2 \times \delta_{[1,2]}$. Liczby mają postać $x = 2.0 + k \cdot 2^{-51}$.
- W przedziale $[0.5, 1]$ ($E = -1$), krok $\delta_z = 2^{-53}$, czyli $\frac{1}{2} \times \delta_{[1,2]}$. Liczby mają postać $x = 0.5 + k \cdot 2^{-53}$.

W **standardzie IEEE 754** odstęp (ULP) między liczbami podwaja się przy każdym przekroczeniu potęgi dwójki. W **obliczeniach maszynowych** oznacza to, że choć **błąd bezwzględny** rośnie wraz z wielkością liczby, **błąd względny** (czyli $\text{ulp}(x)/x$) pozostaje w przybliżeniu stały.

---

4 Zadanie 4: Badanie tożsamości $x \cdot (1/x) = 1$

4.0.1 Opis problemu

Zadanie polegało na (a) znalezieniu eksperymentalnie dowolnej oraz (b) znalezieniu najmniejszej liczby zmiennopozycyjnej x w arytmetyce `Float64` w przedziale $(1, 2)$, dla której tożsamość matematyczna $x \cdot (1/x) = 1$ nie jest spełniona. Oczekiwany błąd to $fl(x \cdot fl(1/x)) \neq 1$.

4.0.2 Opis rozwiązania

Zaimplementowano metodę "brute-force". Skrypt inicjuje $x = 1.0$ i w pętli iteracyjnie przechodzi do następnej liczby maszynowej ($x =$ `nextfloat(x)`), zliczając liczbę kroków k. Dla każdej liczby x oblicza wartość $v = fl(x \cdot fl(1/x))$. Pętla zatrzymuje się, gdy $v \neq 1.0$. Aby umożliwić analizę, program drukuje stan z ostatniej "poprawnej"iteracji ($k - 1$) oraz z pierwszej "błędnej"iteracji (k). Skrypt został uruchomiony z terminala (‘cmd‘), aby zapewnić kompilację JIT i rozsądny czas wykonania.

4.0.3 Wyniki i interpretacja

Eksperyment zakończył się sukcesem, znajdując szukaną wartość po ponad 257 milionach iteracji.

```
--- Zadanie 4(a) i 4(b): Szukanie x w (1, 2) takiego, że x * (1/x) != 1 ---
Rozpoczynam poszukiwania (limit = 300000000 iteracji)...
... Przekroczono 100000000 iteracji, nadal szukam...
... Przekroczono 200000000 iteracji, nadal szukam...

--- Wyniki ---
Znaleziono najmniejszą liczbę x spełniającą warunek (odp. na 4b):

--- Ostatnia 'poprawna' iteracja (k-1) ---
x_(k-1) =               1.00000005722899687e+00
Bitstring x_(k-1):     0011111111110000000000000...0101001
fl(1/x_(k-1)) =               9.99999942771006345e-01
fl(x * fl(1/x)) =     1.00000000000000000e+00
Bitstring wyniku:     0011111111110000000000000...0000000
Wynik == 1.0:         true

--- Pierwsza 'błędna' iteracja (k) ---
x_k =                   1.00000005722899710e+00
Bitstring x_k:         0011111111110000000000000...0101010
Liczba k (x = 1.0 + k*eps(1.0)): 257736490

Sprawdzenie obliczenia dla x_k:
fl(1/x_k) =           9.99999942771006123e-01
fl(x_k * fl(1/x_k)) =   9.99999999999999889e-01
Bitstring wyniku:     0011111111101111111111111...1111111
Wynik != 1.0:         true
```

Interpretacja:

- Pętla zatrzymała się przy $k = 257,736,490$. Jest to odpowiedź na zadanie (b) - jest to najmniejsza liczba kroków $\delta =$ `eps(1.0)`, dla której błąd występuje.
- Odpowiadająca jej liczba $x_k = 1.00000005722899710e + 00$ jest odpowiedzią na zadanie (a) i (b).
- Dla x_{k-1} wynik końcowy był poprawnie zaokrąglony do `1.0`.
- Dla x_k wynik końcowy wyniósł `9.999...889e-01`. Jest to liczba maszynowa bezpośrednio poprzedzająca 1.0, czyli `prevfloat(1.0)`.
- Potwierdza to 'Bitstring wyniku' dla x_k, który ma wykładnik 2^{-1} i mantysę złożoną z samych jedynek, co jest reprezentacją `prevfloat(1.0)`.

4.0.4 Wnioski

Eksperyment dowiódł, że tożsamość $x \cdot (1/x) = 1$ nie jest zachowana w **obliczeniach maszynowych**. Jest to skutek kumulacji błędów zaokrągleń w standardzie **IEEE 754**.

Występują tu dwa błędy:

1. Błąd ϵ_1 przy obliczaniu $fl(1/x)$.

2. Błąd ϵ_2 przy obliczaniu $fl(x \cdot (1/x + \epsilon_1))$.

Dla $k < 257,736,490$ skumulowany błąd był na tyle mały, że ostateczny wynik $1 + \epsilon_{total}$ był nadal bliżej liczby maszynowej 1.0 i był do niej poprawnie zaokrąglany.

W iteracji $k = 257,736,490$ skumulowany błąd ϵ_{total} stał się na tyle duży (i ujemny), że "prawdziwy"wynik $1 - |\epsilon_{total}|$ przekroczył punkt środkowy między `prevfloat(1.0)` a `1.0`, co spowodowało jego zaokrąglenie w dół do `prevfloat(1.0)`, łamiąc tym samym tożsamość.

---

5 Zadanie 5: Błędy sumowania iloczynu skalarnego

5.0.1 Opis problemu

Zadanie polegało na obliczeniu iloczynu skalarnego $S = \sum_{i=1}^{n} x_i y_i$ dla danych wektorów x i y ($n = 5$) przy użyciu czterech różnych algorytmów sumowania: (a) w przód, (b) w tył, (c) od największego do najmniejszego (wg wartości bezwzględnej, z osobnym sumowaniem dodatnich i ujemnych), (d) od najmniejszego do największego (przeciwnie do c). Obliczenia należało wykonać w precyzji `Float32` oraz `Float64` i porównać wyniki z wartością referencyjną $S_{ref} = -1.00657107000000 \cdot 10^{-11}$.

5.0.2 Opis rozwiązania

Zdefiniowano wektory x i y w precyzji `Float64`. Dla każdej badanej precyzji (`Float32`, `Float64`) najpierw obliczono wektor iloczynów $t_i = fl(x_i \cdot y_i)$, konwertując x i y do docelowej precyzji. Następnie zaimplementowano cztery funkcje sumujące wektor t: `alg_a_forward` (pętla $i = 1..n$), `alg_b_backward` (pętla $i = n..1$), `alg_c_largest_to_smallest` (sortowanie dodatnich malejąco, ujemnych rosnąco, sumowanie osobno, dodanie sum częściowych) oraz `alg_d_smallest_to_largest` (sortowanie przeciwne do c). Dla każdego wyniku obliczono błąd względny względem wartości referencyjnej.

5.0.3 Wyniki i interpretacja

Poniżej przedstawiono wyniki uzyskane z uruchomienia skryptu.

```
--- Zadanie 5: Błędy sumowania iloczynu skalarnego ---

========================================
--- Obliczenia dla typu: Float32 ---
========================================
Wektor obliczonych iloczynów (terms = x[i] * y[i]):
terms[1] = +4.040045654296875e+03
terms[2] = -2.759471500000000e+06
terms[3] = -3.164291381835938e+01
terms[4] = +2.755462750000000e+06
terms[5] = +5.570529901888222e-05

--- Wyniki sumowania dla Float32 ---
Metoda           Wynik Obliczony           Błąd Względny
------------------------------------------------------------
(a) W przód      -4.999442994594574e-01     4.967e+10
(b) W tył        -4.543457031250000e-01     4.514e+10
(c) Najw.->Najm. -5.000000000000000e-01     4.967e+10
(d) Najm.->Najw. -5.000000000000000e-01     4.967e+10
```

```
========================================
--- Obliczenia dla typu: Float64 ---
========================================
Wektor obliczonych iloczynów (terms = x[i] * y[i]):
terms[1] = +4.040045551380452e+03
terms[2] = -2.759471276702747e+06
terms[3] = -3.164291531266504e+01
terms[4] = +2.755462874010974e+06
terms[5] = +5.570529967428930e-05

--- Wyniki sumowania dla Float64 ---
Metoda           Wynik Obliczony            Błąd Względny
------------------------------------------------------------
(a) W przód      1.025188136829667e-10      1.118e+01
(b) W tył        -1.564330887049437e-10      1.454e+01
(c) Najw.->Najm. 0.000000000000000e+00      1.000e+00
(d) Najm.->Najw. 0.000000000000000e+00      1.000e+00

========================================
Wartość referencyjna:   -1.006571070000000e-11
```

Interpretacja:

- **Float32:** Wyniki są katastrofalne. Wektor iloczynów 'terms' zawiera dwie bardzo duże liczby o przeciwnych znakach ($t_2 \approx -2.76 \cdot 10^6$ i $t_4 \approx +2.76 \cdot 10^6$). Ich suma powinna być bliska zeru, ale w precyzji `Float32` (ok. 7 cyfr dziesiętnych) dochodzi do katastrofalnej anulacji . Odejmując liczby bliskie sobie co do wartości bezwzględnej, tracimy niemal wszystkie cyfry znaczące. Ostateczne wyniki (≈ -0.5) są rzędy wielkości od wartości referencyjnej ($\approx -10^{-11}$), a błędy względne są gigantyczne ($\approx 10^{10}$). Kolejność sumowania ma niewielki wpływ przy tak dużej utracie precyzji.
- **Float64:** Wyniki są znacznie lepsze, rzędu 10^{-10}, ale nadal obarczone sporym błędem względnym ($\approx 10^1$). Precyzja `Float64` (ok. 16 cyfr dziesiętnych) pozwala na dokładniejsze obliczenie różnicy między dużymi składnikami t_2 i t_4, ale wynik nadal jest bardzo bliski zeru, co uwypukla błędy zaokrągleń pozostałych operacji.
  - Różne wyniki dla metod (a) i (b) potwierdzają, że dodawanie zmiennoprzecinkowe nie jest łączne .
  - Metody (c) i (d), sortujące składniki, dały wynik dokładnie 0.0. Jest to mniej dokładne niż metody (a) i (b), co sugeruje, że sortowanie nie zawsze jest najlepszą strategią, szczególnie przy silnej anulacji.

5.0.4 Wnioski

Eksperyment dobitnie pokazał zjawisko katastrofalnej anulacji w **obliczeniach maszynowych**. Odejmując dwie bliskie sobie, duże liczby, tracimy precyzję. Skutki są druzgocące w niskiej precyzji (`Float32`), ale zauważalne nawet w `Float64`.

Potwierdzono również, że kolejność sumowania ma znaczenie (brak łączności dodawania w arytmetyce zmiennoprzecinkowej), co widać po różnych wynikach metod (a) i (b) dla `Float64`.

Algorytmy sortujące (c, d), często zalecane do minimalizacji błędów przez sumowanie najpierw małych liczb, w tym specyficznym przypadku dały mniej dokładny wynik (0.0) niż proste sumowanie w przód/tył. Podkreśla to, że nie ma uniwersalnie najlepszego algorytmu sumowania, a wybór metody zależy od charakteru danych i potencjalnego występowania anulacji. Zadanie to ilustruje fundamentalne problemy stabilności numerycznej algorytmów.

---

6 Zadanie 6: Stabilność numeryczna $f(x)$ vs $g(x)$

6.0.1 Opis problemu

Zadanie polegało na porównaniu wyników obliczeń dwóch matematycznie równoważnych funkcji:

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

dla malejących wartości argumentu $x = 8^{-k}$ ($k = 1, 2, \dots$) w arytmetyce `Float64`. Celem było zaobserwowanie, jak błędy zaokrągleń wpływają na wyniki i określenie, która forma obliczeń jest bardziej wiarygodna (numerycznie stabilna) dla małych x.

6.0.2 Opis rozwiązania

Zaimplementowano dwie funkcje w Julii, `f(x)` i `g(x)`, które dokładnie odzwierciedlają podane wzory matematyczne. Główna część skryptu iterowała przez $k = 1, 2, \dots, 15$, obliczając $x = 8.0^{-k}$. W każdej iteracji obliczano wartości `f(x)` oraz `g(x)` i drukowano je obok siebie w celu porównania. Pętla zawierała warunek przerwania, gdy wynik `f(x)` stawał się dokładnie zerem, co sygnalizowało utratę precyzji. Wszystkie obliczenia wykonano w standardowej precyzji podwójnej (`Float64`).

6.0.3 Wyniki i interpretacja

Poniżej przedstawiono wyniki uzyskane z uruchomienia skryptu.

```
--- Zadanie 6: Porównanie f(x) = sqrt(x^2+1)-1 i g(x) = x^2/(sqrt(x^2+1)+1) ---
--- Obliczenia w Float64 ---

k        x = 8^(-k)          f(x)                          g(x)
--------------------------------------------------------------------------------
1        1.25000000e-01      7.78221853731864144e-03        7.78221853731870649e-03
2        1.56250000e-02      1.22062862828675733e-04        1.22062862828759014e-04
3        1.95312500e-03      1.90734681382309645e-06        1.90734681382656590e-06
4        2.44140625e-04      2.98023219436061026e-08        2.98023219436061159e-08
5        3.05175781e-05      4.65661287307739258e-10        4.65661287199319041e-10
6        3.81469727e-06      7.27595761418342590e-12        7.27595761415695612e-12
7        4.76837158e-07      1.13686837721616030e-13        1.13686837721609567e-13
8        5.96046448e-08      1.77635683940025046e-15        1.77635683940024889e-15
9        7.45058060e-09      0.00000000000000000e+00        2.77555756156289135e-17

Przerywam: f(x) stało się zerem lub g(x) osiągnęło limit precyzji.
```

Interpretacja:

- Dla dużych wartości x (małe k), wyniki $f(x)$ i $g(x)$ są bardzo zbliżone.

- W miarę jak x maleje (rośnie k), różnica między $f(x)$ a $g(x)$ staje się zauważalna, chociaż obie wartości maleją.
- Przy $k = 8$ ($x \approx 6 \cdot 10^{-8}$), $f(x)$ i $g(x)$ dają wyniki różniące się na ostatnich cyfrach znaczących.
- Przy $k = 9$ ($x \approx 7 \cdot 10^{-9}$), następuje załamanie obliczeń dla $f(x)$. Wynik staje się dokładnie `0.0`. W tym momencie $\sqrt{x^2+1}$ jest tak bliskie 1, że w arytmetyce `Float64` jest zaokrąglane do dokładnie `1.0`. Operacja $1.0 - 1.0$ daje zero. Jest to klasyczny przykład katastrofalnej anulacji (odejmowanie dwóch bardzo bliskich sobie liczb).
- Funkcja $g(x)$ nadal daje poprawny, mały, ale niezerowy wynik ($\approx 2.8 \cdot 10^{-17}$). Forma $g(x)$ unika odejmowania bliskich liczb; zamiast tego wykonuje stabilne numerycznie dodawanie $\sqrt{x^2+1}+1$.

6.0.4 Wnioski

Eksperyment pokazał, że chociaż funkcje $f(x)$ i $g(x)$ są matematycznie równoważne, ich stabilność numeryczna w **obliczeniach maszynowych** jest drastycznie różna dla małych wartości x.

- Wzór $f(x) = \sqrt{x^2+1} - 1$ jest niestabilny numerycznie dla $x \to 0$ z powodu katastrofalnej anulacji . Prowadzi to do całkowitej utraty cyfr znaczących i błędnego wyniku (zero).
- Wzór $g(x) = x^2/(\sqrt{x^2+1}+1)$ jest stabilny numerycznie dla $x \to 0$. Unika on problematycznego odejmowania, zastępując je stabilnym dodawaniem i dzieleniem.

Wyniki obliczeń dla $g(x)$ są wiarygodne w całym przetestowanym zakresie, podczas gdy wyniki dla $f(x)$ stają się bezużyteczne dla $x < 10^{-8}$. To zadanie jest doskonałym przykładem, jak przekształcenie algebraiczne wyrażenia może radykalnie poprawić dokładność obliczeń w arytmetyce zmiennoprzecinkowej zgodnej ze standardem **IEEE 754**.

---

7 Zadanie 7: Błąd aproksymacji pochodnej

7.0.1 Opis problemu

Zadanie polegało na obliczeniu przybliżonej wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ za pomocą wzoru różnicy skończonej w przód:

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

Należało zbadać zachowanie błędu bezwzględnego $|f'(x_0) - \tilde{f}'(x_0)|$ dla malejących kroków $h = 2^{-n}$, gdzie $n = 0, 1, \ldots, 54$. Obliczenia wykonano w arytmetyce `Float64`. Szczególną uwagę należało zwrócić na moment, od którego zmniejszanie h przestaje poprawiać wynik, oraz na zachowanie wartości $1.0 + h$.

7.0.2 Opis rozwiązania

Zaimplementowano funkcję `f(x)` obliczającą $\sin x + \cos 3x$ oraz funkcję `f_prime_exact(x)` obliczającą dokładną pochodną $\cos x - 3 \sin 3x$. Główna część skryptu iterowała przez n od 0 do 54. W każdej iteracji obliczano $h = 2.0^{-n}$, $x_0 + h$, oraz wartość $1.0 + h$. Następnie obliczano przybliżoną pochodną $\tilde{f}'(x_0)$ używając podanego wzoru, dbając o przypadek, gdy $x_0 + h$ zaokrąglało się do x_0 (wtedy wynik aproksymacji staje się 0). Na koniec obliczano i drukowano błąd bezwzględny $|f'(x_0) - \tilde{f}'(x_0)|$.

7.0.3 Wyniki i interpretacja

Poniżej przedstawiono wybrane wyniki uzyskane z uruchomienia skryptu (pełne wyniki w logu).

```
--- Zadanie 7: Błąd aproksymacji pochodnej f(x) = sin(x) + cos(3x) w x0 = 1 ---
--- Obliczenia w Float64 ---

Punkt x0 = 1.0
Dokładna pochodna f'(x0) = 1.16942281688538152e-01

n     h = 2^(-n)        f_approx'(x0)          |f'(x0) - f_approx'(x0)|   1.0 + h (w Float
----------------------------------------------------------------------------------------------
0     1.00000000e+00     2.01798922526859670e+00      1.90104694e+00         2.0000000000
...   ...                ...                          ...                    ...
25    2.98023224e-08     1.16942398250102997e-01      1.16561565e-07         1.0000000298
26    1.49011612e-08     1.16942338645458221e-01      5.69569201e-08         1.0000000149
27    7.45058060e-09     1.16942316293716431e-01      3.46051783e-08         1.0000000074
28    3.72529030e-09     1.16942286491394043e-01      4.80285589e-09  <-- Minimum błędu
29    1.86264515e-09     1.16942226886749268e-01      5.48017889e-08         1.0000000018
30    9.31322575e-10     1.16942167282104492e-01      1.14406434e-07         1.0000000009
...   ...                ...                          ...                    ...
51    4.44089210e-16     0.00000000000000000e+00      1.16942282e-01         1.0000000000
52    2.22044605e-16     -5.00000000000000000e-01     6.16942282e-01         1.0000000000
53    1.11022302e-16     0.00000000000000000e+00      1.16942282e-01         1.0000000000
54    5.55111512e-17     0.00000000000000000e+00      1.16942282e-01         1.0000000000
```

Interpretacja:

- **Początkowa poprawa:** Dla dużych h (małe n), błąd jest duży. W miarę zmniejszania h, błąd maleje. Jest to zgodne z teorią, ponieważ **błąd obcięcia** (truncation error) metody różnicy skończonej jest proporcjonalny do h (dokładniej $O(h)$).
- **Minimum błędu:** Błąd osiąga minimum w okolicach $n = 28$, gdzie $h \approx 3.7 \cdot 10^{-9}$, a błąd wynosi ok. $4.8 \cdot 10^{-9}$.
- **Wzrost błędu:** Dalsze zmniejszanie h (zwiększanie n) powoduje, że błąd **ponownie rośnie**. Jest to spowodowane dominacją **błędu zaokrągleń** (round-off error). W liczniku $\tilde{f}'(x_0)$ odejmujemy dwie bardzo bliskie sobie wartości, $f(x_0 + h)$ i $f(x_0)$, co prowadzi do katastrofalnej anulacji . Dodatkowo, dzielenie przez bardzo małe h wzmacnia ten błąd.
- **Zachowanie $1.0 + h$:** Obserwacja wartości $1.0 + h$ pokazuje, że dla $n = 53$ ($h \approx 1.1 \cdot 10^{-16}$), wartość $1.0 + h$ jest zaokrąglana do dokładnie 1.0. Wtedy $x_0 + h$ staje się równe x_0, licznik aproksymacji pochodnej $f(x_0) - f(x_0)$ staje się zero, a przybliżona pochodna również wynosi zero. Błąd aproksymacji staje się równy wartości bezwzględnej dokładnej pochodnej.
- **Dziwny wynik dla n=52:** Dla $n = 52$, $h = \texttt{eps(1.0)}/2$. Wtedy $1.0 + h$ zaokrągla się do $1.0 + \texttt{eps(1.0)}$, a $x_0 + h$ do $x_0 + \texttt{eps(1.0)}$. Obliczenie $(f(x_0 + \epsilon) - f(x_0))/(\epsilon/2)$ daje bardzo niedokładny wynik.

7.0.4 Wnioski

Eksperyment ilustruje fundamentalny problem w numerycznym różniczkowaniu: wybór optymalnego kroku h.

- Zbyt duże h prowadzi do dużego **błędu obcięcia** metody.
- Zbyt małe h prowadzi do dominacji **błędu zaokrągleń** spowodowanego katastrofalną anulacją w liczniku i wzmocnieniem przez mały mianownik.

Istnieje optymalna wartość h, która minimalizuje sumę tych dwóch błędów. W naszym przypadku dla `Float64` było to $h \approx 10^{-8}$ do 10^{-9}.

Obserwacja $1.0 + h$ pokazała, jak ograniczenia precyzji maszynowej (**standard IEEE 754**) bezpośrednio wpływają na obliczenia: gdy h staje się mniejsze niż $\approx$ `eps(1.0)`$/2$, dodanie go do 1.0 nie daje już oczekiwanego wyniku, co prowadzi do załamania metody różnicowej.
