Sprawozdanie z Laboratorium
Obliczenia Naukowe

Paweł Grzegory

October 26, 2025

1 Zadanie 1: Rozpoznanie arytmetyki

1.1 Wyznaczanie epsilona maszynowego

Epsilon maszynowy (`macheps`) to najmniejsza liczba, która dodana do 1.0 daje wynik większy od 1.0. Poniższy kod w języku Julia wyznacza tę wartość iteracyjnie dla typów `Float16`, `Float32` i `Float64`.

```julia
function find_macheps(T::Type{<:AbstractFloat})
    macheps = T(1.0)
    while T(1.0) + macheps / T(2.0) > T(1.0)
        macheps /= T(2.0)
    end
    return macheps
end
```

Wyniki

```
Porównanie iteracyjnie obliczonego epsilonu maszynowego z funkcją eps():
------------------------------------------------------------
Typ danych          Obliczony Epsilon      eps() z Julii          Różnica
------------------------------------------------------------
Float16             9.765625000000000e-04  9.765625000000000e-04  0.000000000000000e+
Float32             1.192092895507812e-07  1.192092895507812e-07  0.000000000000000e+
Float64             2.220446049250313e-16  2.220446049250313e-16  0.000000000000000e+
------------------------------------------------------------
```

Wnioski Iteracyjnie wyznaczony epsilon maszynowy jest identyczny z wartością zwracaną przez wbudowaną funkcję `eps()`, co potwierdza poprawność implementacji.

Związek z precyzją arytmetyki Liczba zmiennoprzecinkowa w systemie o podstawie β z mantysą o precyzji t cyfr jest reprezentowana jako $x = m \cdot \beta^e$, gdzie mantysa m ma postać $d_0.d_1d_2...d_{t-1}$, a $d_0 \neq 0$.

Liczbę 1.0 można zapisać jako $1.0 \cdot \beta^0$. W tej reprezentacji, mantysa ma wartość 1.00...0. Następna w kolejności, większa liczba maszynowa, ma tę samą wartość wykładnika ($e = 0$), ale najmniejszy możliwy przyrost mantysy. Przyrost ten polega na zwiększeniu ostatniej cyfry mantysy (d_{t-1}). Wartość tej pozycji to $\beta^{-(t-1)}$.

Zatem następna liczba po 1.0 to:

$$1.0 + 1 \cdot \beta^{-(t-1)} \cdot \beta^0 = 1.0 + \beta^{1-t}$$

Epsilon maszynowy, `macheps`, to właśnie różnica między tą liczbą a jedynką:

$$\text{macheps} = (1.0 + \beta^{1-t}) - 1.0 = \beta^{1-t}$$

Precyzja arytmetyki (błąd względny zaokrąglenia) ϵ jest zdefiniowana jako połowa epsilona maszynowego, ponieważ maksymalny błąd zaokrąglenia liczby do najbliższej wartości reprezentowalnej to połowa odległości między sąsiednimi liczbami maszynowymi.

$$\epsilon = \frac{1}{2}\text{macheps} = \frac{1}{2}\beta^{1-t}$$

1.2 Wyznaczanie najmniejszej liczby dodatniej (η)

Liczba η to najmniejsza dodatnia liczba maszynowa. Została wyznaczona przez iteracyjne dzielenie 1.0 przez 2, aż wynik będzie równy 0.

```
function find_eta(T::Type{<:AbstractFloat})
    eta = T(1.0)
    while (eta / T(2.0)) > T(0.0)
        eta = eta / T(2.0)
    end
    return eta
end
```

Wyniki

```
Wyznaczanie najmniejszej dodatniej liczby maszynowej (eta)
------------------------------------------------------------
Analiza dla typu: Float16
  > Wartość wyznaczona iteracyjnie: 5.9604645e-8
  > Wartość z funkcji nextfloat(Float16(0.0)): 5.9604645e-8
  Wyniki są identyczne.
------------------------------------------------------------
Analiza dla typu: Float32
  > Wartość wyznaczona iteracyjnie: 1.4012985e-45
  > Wartość z funkcji nextfloat(Float32(0.0)): 1.4012985e-45
  Wyniki są identyczne.
------------------------------------------------------------
Analiza dla typu: Float64
  > Wartość wyznaczona iteracyjnie: 4.94e-324
  > Wartość z funkcji nextfloat(Float64(0.0)): 4.94e-324
  Wyniki są identyczne.
------------------------------------------------------------
```

Związek z MIN_{sub} Wyznaczona wartość η jest najmniejszą dodatnią liczbą maszynową, czyli najmniejszą liczbą subnormalną (MIN_{sub}).

$$\eta = \text{MIN}_{\text{sub}} = \beta^{c_{\min}-(t-1)}$$

gdzie $c_{\min}$ to najmniejszy wykładnik dla liczb znormalizowanych.

1.3 Funkcja `floatmin` i $\mathrm{MIN}_{\mathrm{nor}}$

Funkcja `floatmin(Type)` zwraca najmniejszą dodatnią **znormalizowaną** liczbę maszynową ($\mathrm{MIN}_{\mathrm{nor}}$).

Typ	Wartość z funkcji `floatmin`
`Float16`	6.104e-5
`Float32`	1.1754944e-38
`Float64`	2.2250738585072014e-308

Związek z $\mathrm{MIN}_{\mathrm{nor}}$ jest bezpośredni:

$$\mathrm{floatmin(Type)} = \mathrm{MIN}_{\mathrm{nor}} = \beta^{c_{\min}}$$

1.4 Wyznaczanie największej liczby maszynowej

Największa liczba maszynowa (`floatmax`) została wyznaczona iteracyjnie. Algorytm składa się z dwóch głównych kroków:

1. **Znalezienie największej potęgi dwójki:** Pętla ‘while‘ mnoży początkową wartość ‘1.0‘ przez ‘2.0‘ tak długo, aż wynik nie stanie się nieskończonością. W ten sposób znajdujemy największą potęgę dwójki, która jest jeszcze reprezentowalna w danym typie. Jest to przybliżona wartość ‘floatmax‘.

2. **Doprecyzowanie wyniku:** Zaczynając od znalezionej potęgi dwójki, algorytm próbuje dodawać do niej coraz mniejsze wartości (połowę poprzedniego kroku, zaczynając od początkowej potęgi). Jeśli dodanie kroku nie powoduje przepełnienia (wynik nie jest nieskończonością), krok jest dodawany do wyniku. Proces ten przypomina wyszukiwanie binarne i pozwala na "wypełnienie" bitów mantysy, aby uzyskać największą możliwą wartość.

Poniższy kod przedstawia implementację tej metody.

```julia
function find_max(T::Type{<:AbstractFloat})
    max_power_of_2 = T(1.0)
    while !isinf(max_power_of_2 * T(2.0))
        max_power_of_2 *= T(2.0)
    end

    max_val = max_power_of_2
    step = max_power_of_2
    while step > T(0.0)
        step /= T(2.0)
        if !isinf(max_val + step)
            max_val += step
        end
    end
    return max_val
end
```

Wyniki

Typ	Wyznaczona największa liczba	Wartość z funkcji `floatmax`
`Float16`	6.55e4	6.55e4
`Float32`	3.4028235e38	3.4028235e38
`Float64`	1.7976931348623157e308	1.7976931348623157e308

Porównanie z typami w C Wartości dla `Float32` i `Float64` odpowiadają maksymalnym wartościom dla typów `float` i `double` w języku C.

Typ w C	Maksymalna wartość
`float`	3.402823e+38
`double`	1.797693e+308

2 Zadanie 2: Epsilon maszynowy metodą Kahana

W. Kahan zauważył, że `macheps` można obliczyć za pomocą wyrażenia $3 \times (4/3 - 1) - 1$. Eksperyment potwierdza tę tezę.

```
function khanEps(T::Type{<:AbstractFloat})
    return T(3.0) * (T(4.0) / T(3.0) - T(1.0)) - T(1.0)
end
```

Wyniki

```
Wyznaczanie epsilonu maszynowego metodą Khana
Typ: Float16, Khan Epsilon: -0.000977, eps() z Julii: 0.000977
Typ: Float32, Khan Epsilon: 1.1920929e-7, eps() z Julii: 1.1920929e-7
Typ: Float64, Khan Epsilon: -2.220446049250313e-16, eps() z Julii: 2.220446049250313e
```

Wnioski Metoda Kahana daje poprawne wyniki, co do wartości bezwzględnej. Dla typu `Float32` zwraca wartość identyczną z `eps()`. Jednak dla typów `Float16` i `Float64` metoda zwraca wartość o tej samej magnitudzie co `macheps`, ale o przeciwnym znaku.

3 Zadanie 3: Rozkład liczb zmiennoprzecinkowych

Eksperyment ma na celu zbadanie rozkładu liczb zmiennoprzecinkowych w standardzie IEEE 754 dla typu `Float64` w różnych przedziałach.

Metoda Do zliczenia liczb zmiennoprzecinkowych w danym przedziale wykorzystano funkcję, która opiera się na właściwościach reprezentacji binarnej IEEE 754.

```
function calculateFloatCount(start::Float64, intervalEnd::Float64)
    if start >= intervalEnd
        return 0
    end
    return reinterpret(UInt64, intervalEnd) - reinterpret(UInt64, start)
end
```

Funkcja `reinterpret(UInt64, x)` traktuje 64-bitowy ciąg bitów liczby `Float64` jako 64-bitową liczbę całkowitą bez znaku. W standardzie IEEE 754 kolejne liczby zmiennoprzecinkowe (dla wartości dodatnich) odpowiadają kolejnym liczbom całkowitym po takiej reinterpretacji. Dzięki temu, odejmując od siebie dwie takie liczby całkowite, uzyskujemy dokładną liczbę "kroków" (reprezentowalnych liczb) między dwiema wartościami zmiennoprzecinkowymi.

Pozostałe wartości w wynikach zostały obliczone w następujący sposób:

- **log2(ilość):** Logarytm o podstawie 2 z liczby maszyn w przedziale. Dla przedziałów postaci $[2^e, 2^{e+1})$, wartość ta wynosi 52, co odpowiada liczbie bitów mantysy w typie `Float64`.
- **Reprezentacja bitowa:** Uzyskana za pomocą funkcji `bitstring()`.
- **Krok:** Logarytm o podstawie 2 z różnicy między dwiema kolejnymi liczbami maszynowymi, np. `log2(nextfloat(1.0) - 1.0)`. Wartość kroku jest stała w obrębie przedziału $[2^e, 2^{e+1})$ i wynosi 2^{e-52}.

```
Ilosc liczb Float64 w [0.5, 1.0): 4503599627370496, log2: 52.0
Reprezentacja 0.5:
   0011111111100000000000000000000000000000000000000000000000000000

Reprezentacja nextfloat(0.5)
   :0011111111100000000000000000000000000000000000000000000000000001

Krok na poczatku: -53.0
Reprezentacja prevfloat(1.0)
   :0011111111101111111111111111111111111111111111111111111111111111

Reprezentacja 1.0:
   0011111111110000000000000000000000000000000000000000000000000000

Krok na koncu:   -53.0

Ilosc liczb Float64 w [1.0, 2.0): 4503599627370496, log2: 52.0
Reprezentacja 1.0:
   0011111111110000000000000000000000000000000000000000000000000000

Reprezentacja nextfloat(1.0)
   :0011111111110000000000000000000000000000000000000000000000000001

Krok na poczatku: -52.0
Reprezentacja prevfloat(2.0)
   :0011111111111111111111111111111111111111111111111111111111111111

Reprezentacja 2.0:
   0100000000000000000000000000000000000000000000000000000000000000

Krok na koncu:   -52.0

Ilosc liczb Float64 w [2.0, 4.0): 4503599627370496, log2: 52.0
```

```
Reprezentacja 2.0:
   0100000000000000000000000000000000000000000000000000000000000000

Reprezentacja nextfloat(2.0)
   :0100000000000000000000000000000000000000000000000000000000000001

Krok na poczatku: -51.0
Reprezentacja prevfloat(4.0)
   :0100000000001111111111111111111111111111111111111111111111111111

Reprezentacja 4.0:
   0100000000010000000000000000000000000000000000000000000000000000

Krok na koncu:   -51.0
```

Wnioski Eksperyment potwierdza, że w każdym z przedziałów postaci $[2^e, 2^{e+1})$ znajduje się taka sama liczba (2^{52}) liczb zmiennoprzecinkowych typu `Float64`. Jednak odległość między kolejnymi liczbami (krok) jest stała tylko w obrębie jednego takiego przedziału.

- **W przedziale [0.5, 1.0)** (dla wykładnika $e = -1$): Krok wynosi $\delta = 2^{-53}$. Każdą liczbę x w tym przedziale można przedstawić jako $x = 0.5 + k \cdot 2^{-53}$, gdzie $k \in \{0, 1, \ldots, 2^{52} - 1\}$.
- **W przedziale [1.0, 2.0)** (dla wykładnika $e = 0$): Krok jest dwukrotnie większy i wynosi $\delta = 2^{-52}$. Każdą liczbę x można przedstawić jako $x = 1.0 + k \cdot 2^{-52}$, gdzie $k \in \{0, 1, \ldots, 2^{52} - 1\}$.
- **W przedziale [2.0, 4.0)** (dla wykładnika $e = 1$): Krok ponownie się podwaja i wynosi $\delta = 2^{-51}$. Każdą liczbę x można przedstawić jako $x = 2.0 + k \cdot 2^{-51}$, gdzie $k \in \{0, 1, \ldots, 2^{52} - 1\}$.

Gęstość rozmieszczenia liczb zmiennoprzecinkowych maleje wraz ze wzrostem ich wartości bezwzględnej. Krok δ podwaja się przy każdym przekroczeniu kolejnej potęgi dwójki.

4 Zadanie 4: Utrata własności arytmetycznych

W arytmetyce zmiennoprzecinkowej operacja mnożenia i dzielenia nie zawsze są do siebie odwrotne. Poniższy kod poszukuje liczby x, dla której $x \times (1/x) \neq 1$.

```julia
function findx(start::Float64, finish::Float64)
    x = start
    while x < finish
        if x * (1.0 / x) != 1.0
            return x
        end
        x = nextfloat(x)
```

```
    end

    return nothing
end
```

Wyniki

```
Znajdowanie liczby x, dla której x * (1/x) != 1
[1,2]: x = 1.000000057228997, wartość x * (1/x) = 0.9999999999999999
Najmniejszy x: x = 5.0e-324, wartość x * (1/x) = Inf
```

Wnioski Znaleziono liczbę x w przedziale $[1, 2]$, dla której operacja $x \times (1/x)$ nie jest równa 1. Wynik '0.9999999999999999' to liczba maszynowa bezpośrednio poprzedzająca '1.0'. Jest to spowodowane błędem zaokrągleń podczas obliczania odwrotności $(1/x)$. Dla najmniejszej dodatniej liczby subnormalnej $x = \eta$, obliczenie jej odwrotności $1/x$ prowadzi do przepełnienia (overflow) do nieskończoności. W rezultacie, wyrażenie $x \times (1/x)$ również daje nieskończoność, co stanowi drastyczne naruszenie własności odwrotności mnożenia.

5 Zadanie 5: Błędy w iloczynie skalarnym

Kolejność wykonywania działań w sumowaniu ma wpływ na wynik w arytmetyce zmiennoprzecinkowej. Iloczyn skalarny dwóch wektorów x i y został obliczony na cztery różne sposoby, aby zbadać ten efekt:

(a) **Sumowanie w przód:** Standardowa metoda, w której iloczyny $x_i y_i$ są sumowane w kolejności od $i = 1$ do n.

(b) **Sumowanie w tył:** Iloczyny są sumowane w odwrotnej kolejności, od $i = n$ do 1.

(c) **Sumowanie od największych:** Najpierw obliczane są wszystkie iloczyny $x_i y_i$. Następnie są one sortowane malejąco według wartości bezwzględnej i sumowane w tej kolejności.

(d) **Sumowanie od najmniejszych:** Podobnie jak w (c), ale iloczyny są sortowane rosnąco według wartości bezwzględnej i sumowane od najmniejszego do największego.

Wyniki

```
--- Obliczenia dla Float32 ---
(a) W przód:          -0.4999443
(b) W tył:            -0.4543457
(c) Od największego:  -0.5
(d) Od najmniejszego: -0.5

--- Obliczenia dla Float64 ---
(a) W przód:          1.0251881368296672e-10
(b) W tył:            -1.5643308870494366e-10
```

```
(c) Od największego:  0.0
(d) Od najmniejszego: 0.0
```

Wnioski Dokładna wartość iloczynu skalarnego wynosi $-1.00657107 \times 10^{-11}$, czyli jest bardzo bliska zeru.

Dla typu `Float64`, który cechuje się wysoką precyzją, metody sortujące (c) i (d) dały wynik zerowy, który jest najbliższy wartości dokładnej. Metody sumowania w przód i w tył wygenerowały niewielkie błędy, ale o przeciwnych znakach, co ilustruje, jak kolejność operacji wpływa na propagację błędów zaokrągleń.

W przypadku typu `Float32` o niższej precyzji, różnice są znacznie bardziej widoczne. Co ciekawe, w tym konkretnym przypadku, standardowe sumowanie w przód i w tył dało wyniki (-0.4999443 i -0.4543457) bliższe dokładnej wartości niż metody sortujące, które obie dały wynik -0.5. Pokazuje to, że choć sortowanie jest ogólnie zalecaną strategią minimalizacji błędów, jego skuteczność zależy od konkretnego rozkładu wartości.

Ogólnie, najlepsze wyniki uzyskuje się, gdy minimalizuje się błędy zaokrągleń. Sumowanie najpierw małych co do wartości bezwzględnej liczb (metoda d) jest dobrą heurystyką, ponieważ zapobiega "pochłonięciu" ich przez liczby o dużej wartości, co jest częstym źródłem błędów w arytmetyce zmiennoprzecinkowej.

6 Zadanie 6: Utrata precyzji w odejmowaniu

Obliczanie wartości dwóch matematycznie równoważnych funkcji $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = x^2/(\sqrt{x^2+1} + 1)$ może dawać różne wyniki dla małych wartości x.

```
function f(x::Float64)
    return sqrt(x^2 + Float64(1.0)) - Float64(1.0)
end

function g(x::Float64)
    return x^2 / (sqrt(x^2 + Float64(1.0)) + Float64(1.0))
end

for i in 1:179
    x = Float64(8.0^-i)
    println("f(8^-$i) = ", f(x), ", g(8^-$i) = ", g(x))
end
```

Wyniki Poniższa tabela przedstawia wartości funkcji $f(x)$ i $g(x)$ dla $x = 8^{-i}$. Ze względu na dużą liczbę wyników, pokazano pierwsze 10 oraz wybrane końcowe iteracje, gdzie różnice stają się najbardziej widoczne.

i	Wartość $f(8^{-i})$	Wartość $g(8^{-i})$
1	7.782218537318641e-03	7.782218537318707e-03
2	1.220628628286757e-04	1.220628628287590e-04
3	1.907346813823097e-06	1.907346813826566e-06
4	2.980232194360610e-08	2.980232194360612e-08
5	4.656612873077393e-10	4.656612871993190e-10
6	7.275957614183426e-12	7.275957614156956e-12
7	1.136868377216160e-13	1.136868377216096e-13
8	1.776356839400250e-15	1.776356839400249e-15
9	0.000000000000000e+00	2.775557561562891e-17
10	0.000000000000000e+00	4.336808689942018e-19
⋮	⋮	⋮
170	0.000000000000000e+00	4.450147717014403e-308
171	0.000000000000000e+00	6.953355807835000e-310
⋮	⋮	⋮
178	0.000000000000000e+00	1.600000000000000e-322
179	0.000000000000000e+00	0.000000000000000e+00

Wnioski Dla małych wartości x (rosnących i), funkcja $f(x)$ cierpi na błąd utraty precyzji. Dzieje się tak, ponieważ dla $x \to 0$, wartość $\sqrt{x^2+1}$ jest bardzo bliska 1, a odejmowanie dwóch bliskich sobie liczb prowadzi do utraty cyfr znaczących. Już dla $i = 9$ ($x \approx 7.45 \times 10^{-9}$), wynik $f(x)$ jest błędnie obliczany jako zero.

Funkcja $g(x)$ jest algebraicznie równoważnym przekształceniem, które eliminuje problematyczne odejmowanie. Dzięki temu zachowuje ona dokładność dla znacznie mniejszych wartości x.

7 Zadanie 7: Błąd w przybliżeniu pochodnej

Przybliżenie pochodnej za pomocą ilorazu różnicowego jest wrażliwe na wybór kroku h. Zbyt małe h prowadzi do błędów zaokrągleń.

```
f(x) = sin(x) + cos(3*x)
df(x) = cos(x) - 3*sin(3*x)

x0 = Float64(1.0)
exact_derivative = df(x0)

for n in 0:54
    h = Float64(2.0^(-n))
    approx_derivative = (f(x0 + h) - f(x0)) / h
    absolute_error = abs(exact_derivative - approx_derivative)
    # ... (reszta kodu do drukowania)
end
```

Wyniki

n	h	Przybliżenie	Dokładna wartość	Błąd
0	1.00e+00	2.017989e+00	1.169423e-01	1.901047e+00
1	5.00e-01	1.870441e+00	1.169423e-01	1.753499e+00
2	2.50e-01	1.107787e+00	1.169423e-01	9.908448e-01
3	1.25e-01	6.232413e-01	1.169423e-01	5.062990e-01
4	6.25e-02	3.704001e-01	1.169423e-01	2.534578e-01
5	3.13e-02	2.434431e-01	1.169423e-01	1.265008e-01
6	1.56e-02	1.800976e-01	1.169423e-01	6.315528e-02
7	7.81e-03	1.484914e-01	1.169423e-01	3.154911e-02
8	3.91e-03	1.327091e-01	1.169423e-01	1.576683e-02
9	1.95e-03	1.248237e-01	1.169423e-01	7.881411e-03
10	9.77e-04	1.208825e-01	1.169423e-01	3.940195e-03
11	4.88e-04	1.189123e-01	1.169423e-01	1.969969e-03
12	2.44e-04	1.179272e-01	1.169423e-01	9.849521e-04
13	1.22e-04	1.174347e-01	1.169423e-01	4.924679e-04
14	6.10e-05	1.171885e-01	1.169423e-01	2.462319e-04
15	3.05e-05	1.170654e-01	1.169423e-01	1.231155e-04
16	1.53e-05	1.170038e-01	1.169423e-01	6.155760e-05
17	7.63e-06	1.169731e-01	1.169423e-01	3.077877e-05
18	3.81e-06	1.169577e-01	1.169423e-01	1.538938e-05
19	1.91e-06	1.169500e-01	1.169423e-01	7.694675e-06
20	9.54e-07	1.169461e-01	1.169423e-01	3.847323e-06
21	4.77e-07	1.169442e-01	1.169423e-01	1.923560e-06
22	2.38e-07	1.169432e-01	1.169423e-01	9.612711e-07
23	1.19e-07	1.169428e-01	1.169423e-01	4.807087e-07
24	5.96e-08	1.169425e-01	1.169423e-01	2.394961e-07
25	2.98e-08	1.169424e-01	1.169423e-01	1.165616e-07
26	1.49e-08	1.169423e-01	1.169423e-01	5.695692e-08
27	7.45e-09	1.169423e-01	1.169423e-01	3.460518e-08
28	3.73e-09	1.169423e-01	1.169423e-01	4.802856e-09
29	1.86e-09	1.169422e-01	1.169423e-01	5.480179e-08
30	9.31e-10	1.169422e-01	1.169423e-01	1.144064e-07
31	4.66e-10	1.169422e-01	1.169423e-01	1.144064e-07
32	2.33e-10	1.169419e-01	1.169423e-01	3.528250e-07
33	1.16e-10	1.169415e-01	1.169423e-01	8.296622e-07
34	5.82e-11	1.169415e-01	1.169423e-01	8.296622e-07
35	2.91e-11	1.169395e-01	1.169423e-01	2.737011e-06
36	1.46e-11	1.169434e-01	1.169423e-01	1.077686e-06
37	7.28e-12	1.169281e-01	1.169423e-01	1.418110e-05
38	3.64e-12	1.169434e-01	1.169423e-01	1.077686e-06
39	1.82e-12	1.168823e-01	1.169423e-01	5.995747e-05
40	9.09e-13	1.168213e-01	1.169423e-01	1.209926e-04
41	4.55e-13	1.169434e-01	1.169423e-01	1.077686e-06
42	2.27e-13	1.166992e-01	1.169423e-01	2.430629e-04
43	1.14e-13	1.162109e-01	1.169423e-01	7.313442e-04
44	5.68e-14	1.171875e-01	1.169423e-01	2.452183e-04
45	2.84e-14	1.132812e-01	1.169423e-01	3.661032e-03
46	1.42e-14	1.093750e-01	1.169423e-01	7.567282e-03
47	7.11e-15	1.093750e-01	1.169423e-01	7.567282e-03
48	3.55e-15	9.375000e-02	1.169423e-01	2.319228e-02
49	1.78e-15	1.250000e-01	1.169423e-01	8.057718e-03
50	8.88e-16	0.000000e+00	1.169423e-01	1.169423e-01
51	4.44e-16	0.000000e+00	1.169423e-01	1.169423e-01
52	2.22e-16	-5.000000e-01	1.169423e-01	6.169423e-01
53	1.11e-16	0.000000e+00	1.169423e-01	1.169423e-01
54	5.55e-17	0.000000e+00	1.169423e-01	1.169423e-01

Błąd absolutny maleje wraz ze zmniejszaniem h do pewnego momentu, a następnie zaczyna rosnąć.

Wnioski Zjawisko to jest wynikiem dwóch przeciwstawnych źródeł błędu:

1. **Błąd metody:** Wynika z przybliżenia pochodnej ilorazem różnicowym.
2. **Błąd zaokrąglenia:** Wynika z utraty precyzji przy odejmowaniu $f(x_0+h)-f(x_0)$, gdy h jest bardzo małe.

Optymalna wartość h to kompromis między tymi dwoma błędami. Zgodnie z tabelą, najmniejszy błąd uzyskano dla $n = 28$ ($h \approx 3.73 \times 10^{-9}$).

Dla h tak małych, że $x_0 + h$ jest równe x_0 w arytmetyce maszynowej (co ma miejsce dla $n \geq 53$), licznik ilorazu staje się zerem. Prowadzi to do całkowicie błędnego wyniku pochodnej równego 0.