Obliczenia Naukowe - Sprawozdanie nr 1

Jędrzej Sajnóg, indeks: 279701

23 października 2025

Spis treści

1 Zadanie 1: Rozpoznanie arytmetyki

1.1 Krótki opis problemu

Zadanie polega na iteracyjnym wyznaczeniu stałych charakteryzujących arytmetykę zmiennoprzecinkową dla typów `Float16`, `Float32` i `Float64` w języku Julia. Należy znaleźć:

- Epsilon maszynowy (`macheps`): najmniejsza liczba > 0 taka, że $1.0 + \text{macheps} > 1.0$.
- Liczbę `eta`: najmniejsza reprezentowalna dodatnia liczba maszynowa.
- Maksymalną wartość (`MAX`): największa skończona liczba maszynowa.

Uzyskane wyniki należy porównać z wartościami zwracanymi przez wbudowane funkcje Julii oraz z wartościami zdefiniowanymi w pliku nagłówkowym języka C (`float.h`).

1.2 Rozwiązanie

Do wyznaczenia wartości zostały zaimplementowane trzy funkcje w języku Julia.

- **Epsilon maszynowy:** Algorytm rozpoczyna od wartości $\text{add} = 1.0$ i w pętli dzieli ją przez 2, dopóki warunek $1.0 + \text{add} > 1.0$ jest spełniony. Ostatnia wartość spełniająca ten warunek jest szukanym epsilonem.
- **Eta:** Algorytm działa podobnie, ale sprawdza warunek $0.0 + \text{add} > 0.0$.
- **MAX:** Algorytm najpierw znajduje największą potęgę dwójki mniejszą od nieskończoności, mnożąc iteracyjnie 1.0 przez 2. Następnie, aby uzyskać największą możliwą wartość, wynik ten jest mnożony przez $(2 - \text{macheps})$, co odpowiada ustawieniu wszystkich bitów mantysy na 1.

1.3 Wyniki i interpretacja

1.3.1 Wyjście programów w Julii

```
# julia zad1_epsilon.jl
Type: Float16, calculated epsilon: 0.000977, true_epsilon: 0.000977
Type: Float32, calculated epsilon: 1.1920929e-7, true_epsilon: 1.1920929e-7
Type: Float64, calculated epsilon: 2.220446049250313e-16, true_epsilon: 2.220446049250313e-16

# julia zad1_eta.jl
Type: Float16, calculated eta: 6.0e-8, true_eta: 6.0e-8
Type: Float32, calculated eta: 1.0e-45, true_eta: 1.0e-45
Type: Float64, calculated eta: 5.0e-324, true_eta: 5.0e-324

# julia zad1_MAX.jl
Type: Float16, calculated max: 6.55e4, true_max: 6.55e4
Type: Float32, calculated max: 3.4028235e38, true_max: 3.4028235e38
Type: Float64, calculated max: 1.7976931348623157e308, true_max: 1.7976931348623157e308
```

1.3.2 Wyjście programu w C

```
# ./a.out
Float32 max: 3.40282346638528859812e+38
Float64 max: 1.79769313486231570815e+308
Float32 eps: 1.19209289550781250000e-07
Float64 eps: 2.22044604925031308085e-16
```

Wartości obliczone iteracyjnie są równe wartościom zwracanym przez funkcje wbudowane w Julii (`eps()`, `nextfloat(zero())`, `floatmax()`). Są one również zgodne ze stałymi zdefiniowanymi w standardzie języka C (`FLT_EPSILON`, `DBL_EPSILON`, `FLT_MAX`, `DBL_MAX`).

Związki między liczbami:

- precyzja arytmetyki ϵ to górne ograniczenie błędu względnego dla danej arytmetyki, `macheps` to najmniejsza wartość taka, że $1+$`macheps` > 1. $\epsilon = \frac{1}{2}$`macheps`
- `eta` to najmniejsza dodatnia liczba subnormalna (`MINsub`), czyli najmniejsza wartość > 0, jaką można zapisać.
- Funkcja `floatmin(T)` zwraca najmniejszą dodatnią liczbę *normalną* (`MINnor`).

1.4 Wnioski

Zaproponowane algorytmy iteracyjne poprawnie wyznaczają kluczowe stałe arytmetyki zmiennoprzecinkowej. Zgodność wyników z wartościami standardowymi potwierdza ich poprawność oraz ilustruje fundamentalne właściwości standardu IEEE 754.

2 Zadanie 2: Wzór Kahana na epsilon maszynowy

2.1 Krótki opis problemu

Zadanie polega na eksperymentalnym sprawdzeniu w języku Julia stwierdzenia Williama Kahana, że epsilon maszynowy można obliczyć za pomocą wyrażenia $3(4/3-1)-1$.

2.2 Rozwiązanie

Obliczenie wyrażenia $3(4/3-1)-1$.

2.3 Wyniki i interpretacja

```
# julia zad2.jl
Type: Float16, true epsilon: 0.000977, approx: -0.000977
Type: Float32, true epsilon: 1.1920929e-7, approx: 1.1920929e-7
Type: Float64, true epsilon: 2.220446049250313e-16, approx: -2.220446049250313e-16
```

Dla typu `Float32` wzór dał dokładną wartość epsilonu. Dla `Float16` i `Float64` wynik jest równy wartości przeciwnej do epsilonu. W zależności od długości mantysy wartość obliczonego może wynosić $+/-\epsilon$

2.4 Wnioski

Wzór Kahana jest poprawną metodą obliczania epsilonu maszynowego.

3 Zadanie 3: Rozmieszczenie liczb zmiennoprzecinkowych

3.1 Krótki opis problemu

Zadanie polega na zbadaniu, jak rozmieszczone są liczby zmiennoprzecinkowe w standardzie IEEE 754 (dla typu `Float64`) w różnych przedziałach: $[1, 2]$, $[1/2, 1]$ oraz $[2, 4]$.

3.2 Rozwiązanie

W przedziale $[1, 2)$ wszystkie liczby mają ten sam wykładnik, a różnią się jedynie 52-bitową mantysą. Oznacza to, że są one rozmieszczone równomiernie, a odległość między dwiema kolejnymi liczbami (krok δ) jest stała i wynosi 2^{-52}. W innych przedziałach, będących potęgami dwójki, krok ten ulega skalowaniu. W przedziale $[2^k, 2^{k+1})$ krok wynosi $2^k \cdot 2^{-52}$. Do analizy został napisany program, który wyświetla reprezentacje binarne kolejnych liczb w danym przedziale, aby zobrazować zmiany w mantysie.

3.3 Wyniki i interpretacja

```
#    julia zad3.jl
example: 0; bitstring: 0011111111100000000000000000000000000000000000000000000000000000; value: 0.5;
example: 1; bitstring: 0011111111100000000000000000000000000000000000000000000000000001; value: 0.5000000000000001;
example: 2; bitstring: 0011111111100000000000000000000000000000000000000000000000000010; value: 0.5000000000000002;
example: 3; bitstring: 0011111111100000000000000000000000000000000000000000000000000011; value: 0.5000000000000003;
example: 4; bitstring: 0011111111100000000000000000000000000000000000000000000000000100; value: 0.5000000000000004;
example: 5; bitstring: 0011111111100000000000000000000000000000000000000000000000000101; value: 0.5000000000000006;
example: 6; bitstring: 0011111111100000000000000000000000000000000000000000000000000110; value: 0.5000000000000007;
example: 7; bitstring: 0011111111100000000000000000000000000000000000000000000000000111; value: 0.5000000000000008;
example: 8; bitstring: 0011111111100000000000000000000000000000000000000000000000001000; value: 0.5000000000000009;
example: 9; bitstring: 0011111111100000000000000000000000000000000000000000000000001001; value: 0.500000000000001;
example: 10; bitstring: 0011111111100000000000000000000000000000000000000000000000001010; value: 0.5000000000000011;
```

Analiza reprezentacji binarnych i wartości liczbowych potwierdza teorię. Widać, że wykładnik (`01111111110`) pozostaje stały, a zmienia się jedynie mantysa, która jest inkrementowana o 1 na najmłodszej pozycji. To prowadzi do równomiernego rozmieszczenia liczb w tym przedziale.

- W przedziale $[1, 2]$ krok $\delta = 2^{-52}$.
- W przedziale $[1/2, 1]$ krok $\delta = 2^{-53}$. Liczby są rozmieszczone dwa razy gęściej.
- W przedziale $[2, 4]$ krok $\delta = 2^{-51}$. Liczby są rozmieszczone dwa razy rzadziej.

Każde przekroczenie potęgi dwójki powoduje zmianę wykładnika, zwiększając wartość kroku δ.

3.4 Wnioski

Liczby zmiennoprzecinkowe nie są rozmieszczone równomiernie na osi liczbowej. Ich gęstość jest największa w okolicy zera i maleje wraz ze wzrostem wartości bezwzględnej.

4 Zadanie 4: Utrata precyzji

4.1 Krótki opis problemu

Zadanie polega na znalezieniu najmniejszej liczby zmiennoprzecinkowej x w przedziale $(1, 2)$ dla typu `Float64`, dla której operacja $x \cdot (1/x)$ nie jest równa 1.

4.2 Rozwiązanie

Problem wynika z błędów zaokrągleń. Obliczenie odwrotności $1/x$ może nie być dokładne w arytmetyce binarnej. Ten błąd, nawet niewielki, po pomnożeniu z powrotem przez x może spowodować, że końcowy wynik będzie różnił się od 1. Algorytm iteruje od 1 w górę z krokiem równym epsilonowi maszynowemu i sprawdza warunek $x \cdot (1/x) \neq 1$.

4.3 Wyniki i interpretacja

```
# julia zad4.jl
smallest a such that a* (1/a) /= 1 is: 1.000000057228997, instead it equals 0.9999999999999999
```

Program znalazł liczbę, dla której operacja nie jest idealnie odwracalna. Wynik 0.9999999999999999 jest bardzo bliski jedności, ale nie jest jej równy w reprezentacji maszynowej. To pokazuje, że podstawowe prawa arytmetyki (np. istnienie elementu odwrotnego) nie zawsze są spełnione w świecie liczb zmiennoprzecinkowych.

4.4 Wnioski

Operacje w arytmetyce zmiennoprzecinkowej są obarczone błędami związanymi z przybliżaniem liczb. Podstawowe prawa arytmetyki (np. istnienie elementu odwrotnego) nie zawsze są spełnione w arytmetyce zmiennoprzecinkowej.

5 Zadanie 5: Obliczanie iloczynu skalarnego

5.1 Krótki opis problemu

Zadanie polega na obliczeniu iloczynu skalarnego dwóch wektorów na cztery różne sposoby, używając pojedynczej (`Float32`) i podwójnej (`Float64`) precyzji. Celem jest zbadanie wpływu kolejności sumowania na wynik.

5.2 Rozwiązanie

Zaimplementowano cztery algorytmy sumowania iloczynów $x_i y_i$:

1. **W przód:** Standardowa pętla od $i = 1$ do n.
2. **W tył:** Pętla od $i = n$ do 1.
3. **Od największego do najmniejszego:** Iloczyny są sortowane malejąco wg wartości bezwzględnej (osobno dodatnie, osobno ujemne), a następnie sumowane.
4. **Od najmniejszego do największego:** Analogicznie, ale sortowane rosnąco.

5.3 Wyniki i interpretacja

```
# julia zad5.jl
========Current type: Float32=========
x: Float32[2.7182817, -3.1415927, 1.4142135, 0.5772157, 0.30103],
y: Float32[1486.2498, 878367.0, -22.37492, 4.7737145f6, 0.000185049]
True sum: -1.0065710699999998e-11
Forward: -0.4999443, : 4.966805766128285e10
Backward: -0.4543457, : 4.5137965580009605e10
Large to small: -0.5, : 4.967359135306108e10
Small to large: -0.5, : 4.967359135306108e10

========Current type: Float64=========
x: [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957],
y: [1486.2497, 878366.9879, -22.37492, 4.773714647e6, 0.000185049]
True sum: -1.0065710699999998e-11
Forward: 1.0251881368296672e-10, : 11.184955313981629
Backward: -1.5643308870494366e-10, : 14.541186645165917
Large to small: 0.0, : 1.0
Small to large: 0.0, : 1.0
```

Dla typu `Float32`, wszystkie metody dały całkowicie błędne wyniki. Ograniczona precyzja nie jest w stanie poradzić sobie z dużymi różnicami w rzędach wielkości iloczynów.

Dla typu `Float64`, wyniki są znacznie lepsze, ale wciąż niedokładne. Różnice między sumowaniem "w przód"i "w tył"pokazują, że dodawanie zmiennoprzecinkowe nie jest łączne. Prawidłowy wynik jest bardzo bliski zeru. Problem polega na tym, że w sumie występują bardzo duże wartości dodatnie i ujemne, które niemal się znoszą. Tego typu zjawisko często w literaturze nazywane jest katastrofalną anihilacją.

5.4 Wnioski

Kolejność operacji arytmetycznych ma znaczny wpływ na dokładność wyniku. W tym konkretnym przypadku, ze względu na katastrofalną anihilację, żadna z prostych metod sumowania nie daje zadowalającego rezultatu, nawet w podwójnej precyzji.

6 Zadanie 6: Numeryczna stabilność wzorów

6.1 Krótki opis problemu

Zadanie polega na porównaniu dwóch matematycznie równoważnych funkcji dla małych wartości argumentu x i ocenie, która z nich jest bardziej wiarygodna w obliczeniach numerycznych.

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

6.2 Rozwiązanie

Wzór na $f(x)$ jest podatny na błędy numeryczne dla małych x. W takiej sytuacji $x^2 + 1$ jest liczbą bardzo bliską 1, a $\sqrt{x^2+1}$ jest jeszcze bliższe 1. Odejmując od niej 1, dokonujemy odejmowania dwóch bardzo bliskich sobie liczb, co prowadzi utraty precyzji. Wzór na $g(x)$matematycznie równoważny $f(x)$ nie ma tej wady, ponieważ nie występuje w nim problematyczne odejmowanie.

6.3 Wyniki i interpretacja

Tabela przedstawia wybrane wyniki dla malejących wartości $x = 8^{-i}$.

i	**x**	**f(x)**	**g(x)**
1	1.25×10^{-1}	$7.782\,218\,537\,318\,641\,4 \times 10^{-3}$	$7.782\,218\,537\,318\,706\,5 \times 10^{-3}$
2	1.5625×10^{-2}	$1.220\,628\,628\,286\,757\,3 \times 10^{-4}$	$1.220\,628\,628\,287\,590\,1 \times 10^{-4}$
3	$1.953\,125 \times 10^{-3}$	$1.907\,346\,813\,823\,096\,5 \times 10^{-6}$	$1.907\,346\,813\,826\,566 \times 10^{-6}$
4	$2.441\,406\,25 \times 10^{-4}$	$2.980\,232\,194\,360\,610\,3 \times 10^{-8}$	$2.980\,232\,194\,360\,611\,6 \times 10^{-8}$
5	$3.051\,757\,812\,5 \times 10^{-5}$	$4.656\,612\,873\,077\,393 \times 10^{-10}$	$4.656\,612\,871\,993\,190\,4 \times 10^{-10}$
6	$3.814\,697\,265\,625 \times 10^{-6}$	$7.275\,957\,614\,183\,426 \times 10^{-12}$	$7.275\,957\,614\,156\,956 \times 10^{-12}$
7	$4.768\,371\,582\,031\,25 \times 10^{-7}$	$1.136\,868\,377\,216\,160\,3 \times 10^{-13}$	$1.136\,868\,377\,216\,095\,7 \times 10^{-13}$
8	$5.960\,464\,477\,539\,063 \times 10^{-8}$	$1.776\,356\,839\,400\,250\,5 \times 10^{-15}$	$1.776\,356\,839\,400\,248\,9 \times 10^{-15}$
		f(x) zaczyna zwracać 0	
9	$7.450\,580\,596\,923\,828 \times 10^{-9}$	0.0	$2.775\,557\,561\,562\,891\,4 \times 10^{-17}$
10	$9.313\,225\,746\,154\,785 \times 10^{-10}$	0.0	$4.336\,808\,689\,942\,018 \times 10^{-19}$
		...	
98	$2.010\,764\,683\,385\,949 \times 10^{-87}$	0.0	$2.021\,587\,305\,976\,097\,5 \times 10^{-174}$
99	$2.513\,455\,854\,232\,436 \times 10^{-88}$	0.0	$3.158\,730\,165\,587\,652\,3 \times 10^{-176}$
100	$3.141\,819\,817\,790\,545 \times 10^{-89}$	0.0	$4.935\,515\,883\,730\,707 \times 10^{-178}$

Wyniki pokazują, że wraz ze spadkiem wartości x, funkcja $f(x)$ szybko traci precyzję i od $i = 9$ ($x \approx 7.45 \cdot 10^{-9}$) zwraca 0. W tym samym czasie funkcja $g(x)$ nadal oblicza poprawne, małe, niezerowe wartości.

6.4 Wnioski

Matematycznie równoważne wyrażenia mogą mieć drastycznie różną stabilność numeryczną. Wyniki obliczeń dla funkcji $g(x)$ są wiarygodne, podczas gdy dla $f(x)$ nie. Odejmowanie bliskich sobie liczb w algorytmach może prowadzić do niestabilności numerycznej.

7 Zadanie 7: Różniczkowanie numeryczne

7.1 Krótki opis problemu

Zadanie polega na obliczeniu przybliżonej wartości pochodnej funkcji $f(x) = \sin(x)+\cos(3x)$ w punkcie $x_0 = 1$ przy użyciu wzoru różnicy progresywnej: $f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$. Należy zbadać, jak zmienia się błąd przybliżenia w zależności od malejącej wartości kroku $h = 2^{-n}$.

7.2 Rozwiązanie

Wyznaczamy dokładną wartość pochodnej:

$$f'(x_0) = cos(x) - 3sin(3x)$$

i porównujemy ją z iteracyjnie wyznaczanym przybliżeniem.

7.3 Wyniki i interpretacja

Prawdziwa wartość pochodnej: $f'(1) \approx 0.11694228168853815$. Tabela przedstawia wyniki dla n od 0 do 54.

n	**1+h**	**Przyblizenie f'(x_0)**	**Blad bezwzgledny**
0	2.0	2.017 989 225 268 596 7	1.901 046 943 580 058 5
1	1.5	1.870 441 397 931 647 2	1.753 499 116 243 109
2	1.25	1.107 787 095 234 297 4	9.908 448 135 457 593 $\times 10^{-1}$
3	1.125	6.232 412 792 975 817 $\times 10^{-1}$	5.062 989 976 090 435 $\times 10^{-1}$
4	1.0625	3.704 000 662 035 192 $\times 10^{-1}$	2.534 577 845 149 81 $\times 10^{-1}$
5	1.031 25	2.434 430 743 975 468 7 $\times 10^{-1}$	1.265 007 927 090 087 $\times 10^{-1}$
6	1.015 625	1.800 975 633 073 278 5 $\times 10^{-1}$	6.315 528 161 878 97 $\times 10^{-2}$
7	1.007 812 5	1.484 913 953 710 958 $\times 10^{-1}$	3.154 911 368 255 764 $\times 10^{-2}$
8	1.003 906 25	1.327 091 142 805 159 $\times 10^{-1}$	1.576 683 259 197 775 3 $\times 10^{-2}$
9	1.001 953 125	1.248 236 929 407 085 $\times 10^{-1}$	7.881 411 252 170 345 $\times 10^{-3}$
10	1.000 976 562 5	1.208 824 768 110 616 8 $\times 10^{-1}$	3.940 195 122 523 526 5 $\times 10^{-3}$
11	1.000 488 281 25	1.189 122 504 688 384 7 $\times 10^{-1}$	1.969 968 780 300 313 $\times 10^{-3}$
12	1.000 244 140 625	1.179 272 337 390 102 6 $\times 10^{-1}$	9.849 520 504 721 099 $\times 10^{-4}$
13	1.000 122 070 312 5	1.174 347 496 107 657 2 $\times 10^{-1}$	4.924 679 222 275 685 $\times 10^{-4}$
14	1.000 061 035 156 25	1.171 885 136 209 311 9 $\times 10^{-1}$	2.462 319 323 930 373 $\times 10^{-4}$
15	1.000 030 517 578 125	1.170 653 971 457 795 7 $\times 10^{-1}$	1.231 154 572 414 183 7 $\times 10^{-4}$
16	1.000 015 258 789 062 5	1.170 038 392 883 725 5 $\times 10^{-1}$	6.155 759 983 439 424 $\times 10^{-5}$
17	1.000 007 629 394 531 2	1.169 730 604 597 134 5 $\times 10^{-1}$	3.077 877 117 529 937 $\times 10^{-5}$
18	1.000 003 814 697 265 6	1.169 576 710 672 117 8 $\times 10^{-1}$	1.538 937 867 362 477 6 $\times 10^{-5}$
19	1.000 001 907 348 632 8	1.169 499 763 636 849 8 $\times 10^{-1}$	7.694 675 146 829 866 $\times 10^{-6}$
20	1.000 000 953 674 316 4	1.169 461 290 119 215 8 $\times 10^{-1}$	3.847 323 383 432 410 5 $\times 10^{-6}$
21	1.000 000 476 837 158 2	1.169 442 052 487 284 $\times 10^{-1}$	1.923 560 190 242 312 7 $\times 10^{-6}$
22	1.000 000 238 418 579	1.169 432 429 596 781 7 $\times 10^{-1}$	9.612 711 400 208 696 $\times 10^{-7}$
23	1.000 000 119 209 289 6	1.169 427 623 972 296 7 $\times 10^{-1}$	4.807 086 915 192 826 $\times 10^{-7}$
24	1.000 000 059 604 644 8	1.169 425 211 846 828 5 $\times 10^{-1}$	2.394 961 446 938 737 $\times 10^{-7}$
25	1.000 000 029 802 322 4	1.169 423 982 501 03 $\times 10^{-1}$	1.165 615 648 446 305 4 $\times 10^{-7}$
26	1.000 000 014 901 161 2	1.169 423 386 454 582 2 $\times 10^{-1}$	5.695 692 006 923 991 4 $\times 10^{-8}$
27	1.000 000 007 450 580 6	1.169 423 162 937 164 3 $\times 10^{-1}$	3.460 517 827 846 843 $\times 10^{-8}$
28	1.000 000 003 725 290 3	1.169 422 864 913 940 4 $\times 10^{-1}$	4.802 855 890 773 117 $\times 10^{-9}$
Błąd zaczyna rosnąć z powodu błędów zaokrąglenia			
29	1.000 000 001 862 645 1	1.169 422 268 867 492 7 $\times 10^{-1}$	5.480 178 888 461 751 $\times 10^{-8}$
30	1.000 000 000 931 322 6	1.169 421 672 821 044 9 $\times 10^{-1}$	1.144 064 336 600 081 3 $\times 10^{-7}$

n	1+h	Przyblizenie f'(x_0)	Blad bezwzgledny
31	1.000 000 000 465 661 3	$1.169\,421\,672\,821\,044\,9\times10^{-1}$	$1.144\,064\,336\,600\,081\,3\times10^{-7}$
32	1.000 000 000 232 830 6	$1.169\,419\,288\,635\,253\,9\times10^{-1}$	$3.528\,250\,127\,615\,706\,3\times10^{-7}$
33	1.000 000 000 116 415 3	$1.169\,414\,520\,263\,671\,9\times10^{-1}$	$8.296\,621\,709\,646\,956\times10^{-7}$
34	1.000 000 000 058 207 7	$1.169\,414\,520\,263\,671\,9\times10^{-1}$	$8.296\,621\,709\,646\,956\times10^{-7}$
35	1.000 000 000 029 103 8	$1.169\,395\,446\,777\,343\,8\times10^{-1}$	$2.737\,010\,803\,777\,195\,6\times10^{-6}$
36	1.000 000 000 014 552	$1.169\,433\,593\,75\times10^{-1}$	$1.077\,686\,461\,847\,804\,4\times10^{-6}$
37	1.000 000 000 007 276	$1.169\,281\,005\,859\,375\times10^{-1}$	$1.418\,110\,260\,065\,219\,6\times10^{-5}$
38	1.000 000 000 003 638	$1.169\,433\,593\,75\times10^{-1}$	$1.077\,686\,461\,847\,804\,4\times10^{-6}$
39	1.000 000 000 001 819	$1.168\,823\,242\,187\,5\times10^{-1}$	$5.995\,746\,978\,815\,219\,6\times10^{-5}$
40	1.000 000 000 000 909 5	$1.168\,212\,890\,625\times10^{-1}$	$1.209\,926\,260\,381\,522\times10^{-4}$
41	1.000 000 000 000 454 7	$1.169\,433\,593\,75\times10^{-1}$	$1.077\,686\,461\,847\,804\,4\times10^{-6}$
42	1.000 000 000 000 227 4	$1.166\,992\,187\,5\times10^{-1}$	$2.430\,629\,385\,381\,522\times10^{-4}$
43	1.000 000 000 000 113 7	$1.162\,109\,375\times10^{-1}$	$7.313\,441\,885\,381\,522\times10^{-4}$
44	1.000 000 000 000 056 8	$1.171\,875\times10^{-1}$	$2.452\,183\,114\,618\,478\times10^{-4}$
45	1.000 000 000 000 028 4	$1.132\,812\,5\times10^{-1}$	$3.661\,031\,688\,538\,152\times10^{-3}$
46	1.000 000 000 000 014 2	$1.093\,75\times10^{-1}$	$7.567\,281\,688\,538\,152\times10^{-3}$
47	1.000 000 000 000 007	$1.093\,75\times10^{-1}$	$7.567\,281\,688\,538\,152\times10^{-3}$
48	1.000 000 000 000 003 6	9.375×10^{-2}	$2.319\,228\,168\,853\,815\,2\times10^{-2}$
49	1.000 000 000 000 001 8	1.25×10^{-1}	$8.057\,718\,311\,461\,848\times10^{-3}$
50	1.000 000 000 000 000 9	0.0	$1.169\,422\,816\,885\,381\,5\times10^{-1}$
51	1.000 000 000 000 000 4	0.0	$1.169\,422\,816\,885\,381\,5\times10^{-1}$
52	1.000 000 000 000 000 2	-5.0×10^{-1}	$6.169\,422\,816\,885\,382\times10^{-1}$
53	1.0	0.0	$1.169\,422\,816\,885\,381\,5\times10^{-1}$
54	1.0	0.0	$1.169\,422\,816\,885\,381\,5\times10^{-1}$

Początkowo błąd maleje wraz ze zmniejszaniem h. Najmniejszy błąd został osiągnięty dla $n = 28$ ($h \approx 3.7 \cdot 10^{-9}$). Dalsze zmniejszanie h powoduje wzrost błędu. Gdy h staje się tak małe ($n = 53$), że $1 + h$ jest reprezentowane w maszynie jako 1, licznik staje się zerem, a przybliżenie pochodnej jest błędnie równe 0.

7.4 Wnioski

Wybór kroku h w metodach różnic skończonych jest nietrywialny. Istnieje optymalna wartość h, która minimalizuje błąd przybliżenia. Zarówno zbyt duży, jak i zbyt mały krok prowadzą do niedokładnych wyników.