Obliczenia naukowe - sprawozdanie z listy 1

Mikołaj Janusz

23.10.2022

1 Zadanie 1 - rozpoznanie arytmetyki

1.1 Opis zadania

Celem zadania jest rozpoznanie arytmetyki float za pomocą programów napisanych w języku Julia. W tym celu należało wyznaczyć iteracyjnie następujące wartości:

- *epsilon maszynowy* - najmniejszą liczbę dodatnią x taką, że $fl(1.0 + x) > 1$
- *eta* - najmniejsza liczba dodatnia reprezentowana w zadanej arytmetyce
- *MAX* - największa liczba dodatnia reprezentowana w zadanej arytmetyce

1.2 Macheps - opis rozwiązania

1.2.1 Opis algorytmu

Epsilon maszynowy został wyznaczony poprzez iteracyjne dzielenie liczby 1 przez 2 do momentu, w którym uzyskana zostanie równość. W warunku pętli sprawdzamy nierówność dla wartości zmiennej y podzielonej przez 2, dzięki czemu unikniemy końcowego mnożenia zmiennej y razy 2.

```
function macheps(type)
    y = type(1);
    while type(1.0 + 0.5 * y) > type(1.0)
        y = type(0.5 * y)
    end
    return y;
end
```

1.2.2 Analiza wyników

Wyniki uzyskane za pomocą przedstawionego algorytmu są zaprezentowane w poniższej tabeli (zapis 2.220446049250313e-16 w systemie dziesiętnym oznacza liczbę $2.220446049250313 \cdot 10^{-16}$) :

Arytmetyka	Wartość zwracana przez algorytm	Wartość zwracana przez funckję eps	Wartość z pliku nagłówkowego float.h
half (Float16)	0.000977	0.000977	brak odpowiednika
single (Float32)	1.1920929e-7	1.1920929e-7	1.19209e-7
double (Float64)	2.220446049250313e-16	2.220446049250313e-16	2.22045e-16

Tabela 1: Porównanie wartości zwracanych przez algorytm wyznaczający epsilon maszynowy z wartościami zwracanymi przez inne funkcje.

Wartości wyznaczone przez algorytm są równe wartościom zwracanym przez funkcję eps, a wartości zawarte w pliku nagłówkowym biblioteki float.h nieznacznie odbiegają od tych wyznaczonych w języku Julia (dla arytmetyki half nie ma odpowiednika w języku C).

1.2.3 Związek machepsa z precyzją arytmetyki

Liczbę $\epsilon = \frac{1}{2}\beta^{1-t}$ nazywamy precyzją arytmetyki. Zauważmy, że $rd(x) = x(1+\delta)$, gdzie $\delta \leq \epsilon$. Dla $x = 1$ równanie to upraszcza się do $rd(1) = 1 + \delta$. Równanie to będzie spełnione dla $\delta \leq \epsilon$, a więc ϵ to największa taka liczba dodatnia, dla której ta równość jest spełniona. Ponadto:

dla arytmetyki half: $\epsilon = \frac{1}{2} \cdot 2^{1-10} = 2^{-10}$
dla arytmetyki single: $\epsilon = \frac{1}{2} \cdot 2^{1-23} = 2^{-23}$
dla arytmetyki double: $\epsilon = \frac{1}{2} \cdot 2^{1-52} = 2^{-52}$

Gdy otrzymane wyżej wyniki wpiszemy do interpretera języka Julia, otrzymamy wyniki bardzo bliskie wyznaczonym wcześniej machepsom:

```
2^(-10) = 0.0009765625
2^(-23) = 1.1920928955078125e-7
2^(-52) = 2.220446049250313e-16
```

1.3 Eta - opis rozwiązania

1.3.1 Opis algorytmu

Eta została wyznaczona poprzez iteracyjne dzielenie liczby 1 przez 2 do momentu, w którym zmienna pomocnicza będzie równa 0.

```
function myeta(type)
    y = type(1.0);
    while type(y * 0.5) > 0
        y = type(y * 0.5)
    end
    return y
end
```

Podobnie jak w przypadku wyznaczania machepsa, pomocne jest sprawdzanie warunku już dla następnego dzielenia. Dzięki temu nie musimy przechowywać w pamięci poprzedniej wartości zmiennej pomocniczej.

1.3.2 Analiza wyników

Arytmetyka	Wartość zwracana przez algorytm	Wartość zwracana przez funkcję nextfloat
half (Float16)	6.0e-8	6.0e-8
single (Float32)	1.0e-45	1.0e-45
double (Float64)	5.0e-324	5.0e-324

Tabela 2: Porównanie wartości zwracanych przez algorytm wyznaczający liczbę eta z wartością zwracaną przez funkcję nextfloat

Wartości zwracane przez funkcję nextfloat są równe wartościom zwracanym przez algorytm dla wszystkich badanych typów arytmetyk.

1.3.3 Związek liczby eta z MIN_{sub}

MIN_{sub} to najmniejsza liczba dodatnia, którą można wyrazić w danej arytmetyce, która nie musi być w postaci znormalizowanej. Wyraża się ona wzorem:

$$\text{MIN}_{\text{sub}} = 2^{1-t} \cdot 2^{c}$$

$c = -2^{d-1} + 2$
d - liczba bitów przeznaczona na zapis cechy,
t - liczba bitów przeznaczona na zapis mantysy.

Wartości tej liczby dla zadanych arytmetyk wynoszą:
dla arytmetyki half: $2^{-10} \cdot 2^{-14} = 2^{24}$
dla arytmetyki single: $2^{-23} \cdot 2^{-126} = 2^{-149}$
dla arytmetyki double: $2^{-52} \cdot 2^{-1022} = 2^{-1074}$

Za pomocą interpretera języka Julia możemy wyznaczyć te wartości:

```
Float16(2^(-24)) = 6.0e-8
Float32(2^(-149)) = 1.0e-45
Float64(2^(-1074)) = 5.0e-324
```

Wartości te są równe wartościom zwracanym przez funkcję eta.

1.3.4 Związek wartości funkcji floatmin z wartością MIN_{nor}

MIN_{nor} - najmniejsza znormalizowana liczba dodatnia w danej arytmetyce. Wyrażana jest wzorem:

$$\text{MIN}_{\text{nor}} = 2^{c}$$

$c = -2^{d-1} + 2$
d - liczba bitów przeznaczona na zapis cechy
t - liczba bitów przeznaczona na zapis mantysy.

Wartości tej liczby dla zadanych arytmetyk wynoszą:

dla arytmetyki half: 2^{-14}
dla arytmetyki single: 2^{-126}
dla arytmetyki double: 2^{-1022}

Za pomocą języka Julia możemy obliczyć te wartości:

```
Float16(2^(-14)) = Float16(6.104e-5)
Float32(2^(-126)) = 1.1754944e-38
Float64(2^(-1022)) = 2.2250738585072014e-308
```

Wartości funkcji floatmin dla kolejnych arytmetyk wynoszą:

```
floatmin(Float16) = 6.104e-5
floatmin(Float32) = 1.1754944e-38
floatmin(Float64) = 2.2250738585072014e-308
```

Wartości te są równe wartościom MIN_{nor} dla wszystkich trzech badanych arytmetyk.

1.4 MAX - opis rozwiązania

Do wyznaczenia liczby MAX - największej liczby dodatniej reprezentowanej w danej arytmetyce posłużymy się następującą funkcją:

```
function myMAX(type)
    y = prevfloat(type(1.0))
    while(isinf(y)==false)
        y = 2*y;
    end
    return y;
end
```

Obliczenia rozpoczynamy od liczby prevfloat(1.0), która jest sąsiadującą z lewej strony liczbą maszynową dla liczby 1. Korzystając z funkcji bitstring możemy zobaczyć jej zapis bitowy (przykład dla arytmetyki half):

```
bitstring(prevfloat(Float16(1.0))) = 0011101111111111
```

Każde kolejne mnożenie będzie oznaczało powiększenie cechy, ale mantysę będzie zawsze stanowił ciąg 10 jedynek. Maksymalną możliwa liczba będzie uzyskana, gdy cecha przekroczy dozwolony rozmiar. Dlatego wartość początkowa zmiennej y jest poprawna.

Arytmetyka	Wartość zwracana przez algorytm	Wartość zwracana przez funckję floatmax	Wartość z pliku nagłówkowego float.h
half (Float16)	6.55e4	6.55e4	brak odpowiednika
single (Float32)	3.4028235e38	3.4028235e38	3.40282e38
double (Float64)	1.7976931348623157e308	1.7976931348623157e308	1.79769e308

Tabela 3: Porównanie wartości zwracanych przez algorytm wyznaczający liczbę MAX z wartościami zwracanymi przez inne funkcje

Podobnie jak w punkcie 1.2, wartości zwracane przez algorytm i wartości zwracane przez funkcję floatmax dla odpowiednich typów się nie różnią. Wartości te nieznacznie odbiegają od wartości odpowiednika tych funkcji w języku C.

1.5 Wnioski

1. Algorytmy zostały zaimplementowane poprawnie (co wykazuje zgodność wartości zwracanych przez zaimplementowane funkcje z wartościami zwracanymi przez funkcje wbudowane w języku Julia)

2. Obliczenia prowadzone na dowolnym komputerze mają swoje ograniczenia (co uzasadnia istnienie zer maszynowych czy liczb eta). Nawet rachunki wykonywane w arytmetyce double są narażone na wyniki niezgodne z matematycznymi prawami arytmetycznymi.

3. Co prawda prowadząc obliczenia na komputerze istnieje ryzyko błędów i niedokładności, lecz jesteśmy w stanie dosyć dokładnie opisać te niedokładności (m.in za pomocą precyzji arytmetyki).

2 Zadanie 2 - badanie innego sposobu wyznaczania machepsa

2.1 Opis zadania

Zadanie polega na zbadaniu słuszności stwierdzenia, że da się wyznaczyć epsilon maszynowy poprzez obliczenie wyrażenia $3 \cdot (\frac{4}{3} - 1) - 1$ w zadanej arytmetyce.

2.2 Opis rozwiązania

Do rozstrzygnięcia słuszności tej hipotezy posłuży poniższa funkcja zaimplementowana w języku Julia:

```
function zad2(type)
    return type(type(3) * ((type(4)/type(3))-type(1))-type(1))
end
```

2.2.1 Analiza wyników

Wartości tej funkcji są przedstawione poniżej:

Arytmetyka	Wartość zwracana przez funkcję	Epsilon maszynowy
half (Float16)	−0.000977	0.000977
single (Float32)	1.1920929e-7	1.1920929e-7
double (Float64)	2.220446049250313e-16	2.220446049250313e-16

Tabela 4: Porównanie wartości funkcji dla różnych typów wraz z wartościami epsilona maszynowego dla tych typów

Wyniki otrzymane poprzez obliczenie wartości tego wyrażenia są zgodne z epsilonem maszynowym co do wartości bezwzględnej.

2.2.2 Wnioski

Idea obliczania epsilona maszynowego zaproponowana w tym zadaniu jest dosyć skuteczna - w każdej z badanych arytemtyk wartości były prawidłowe co do wartości bezwzględnej. Nie daje ona jednak wyników w pełni poprawnych.

3 Zadanie 3 - rozmieszczenie liczb zmiennopozycyjnych

3.1 Opis zadania

Zadanie polegało na sprawdzeniu, że liczby zmiennopozycyjne w arytmetyce double zgodnej ze standardem IEEE754 (Float64 w języku Julia) są w przedziale $[1, 2)$ równomiernie rozmieszczone z krokiem $\delta = 2^{-52}$ oraz wyznaczyć rozmieszczenie tych liczb w przedziałach $[\frac{1}{2}, 1)$ oraz $[2, 4)$

3.2 Opis rozwiązania

3.2.1 Sprawdzanie długości kroku z wykorzystaniem własności standardu IEEE754 i arytmetyki zmiennopozycyjnej

Zgodnie ze standardem IEEE754, liczby są równomiernie rozmieszczone w danym przedziale, jeśli każda liczba z tego przedziału ma taki sam wykładnik, a różni się jedynie mantysą. Dlatego też liczby w przedziałach $[1, 2)$, $[\frac{1}{2}, 1)$ oraz $[2, 4)$ będą równomiernie rozmieszczone, ponieważ każda z nich ma wykładniki odpowiednio: 0111111111, 01111111111 oraz 10000000000. Odległość między dwoma sąsiadującymi liczbami musi być równa:

$$2^{exp-1023} \cdot 2^{-52}.$$

Funkcja **zad3standard**(a,b,step) przyjmuje argumenty w postaci początku i końca przedziału $[a, b)$ oraz krok *step* i sprawdza, czy liczby są równomiernie rozmieszczone z podanym krokiem.

```
function zad3standard(startNumber, stopNumber, stepNumber)
    stopNumber = prevfloat(stopNumber)
    startNumberExponent = SubString(bitstring(startNumber),2:12)
    stopNumberExponent = SubString(bitstring(stopNumber),2:12)
    if(startNumberExponent!=stopNumberExponent)
        return false
    else
        return ((2.0^(parse(Int, startNumberExponent, base = 2)
        - 1023))*2.0^(-52) == stepNumber)
    end
end
```

3.2.2 Sprawdzanie każdego kroku po kolei

Drugie, bardziej żmudne i czasochłonne podejście to sprawdzenie długości kroków między kolejnymi liczbami zmiennoprzecinkowymi. Funkcja **zad3**(a,b,step) sprawdza kolejne odległości między liczbami zmiennoprzecinkowymi i jeśli natrafi na odległość nierówną wartości step, to zwraca false. Ponieważ dla arytmetyki double mamy 2^{52} różnych mantys dla ustalonej eksponenty, czas wykonywania tej funkcji dla przedziału $[1, 2)$ i prawidłowego kroku byłby bardzo duży.

```
function zad3(startNumber, stopNumber, stepNumber)
    i = Int(0)
    while(startNumber!=stopNumber)
        if(checkDistance(startNumber)!=stepNumber)
            return false
        end
        startNumber = nextfloat(startNumber)
        i = i+1
    end
    return true
end
```

3.3 Analiza wyników

Wartości kroku dla kolejnych przedziałów wynikające z własności arytmetyki zmiennopozycyjnej zostały przedstawione w poniższej tabeli i sprawdzone przez funkcję zad2standard:

przedział	długość kroku
$[\frac{1}{2}, 1)$	1.1102230246251565e-16
$[1, 2)$	2.220446049250313e-16
$[2, 4)$	4.440892098500626e-16

Tabela 5: Długość kroku dla kolejnych przedziałów zweryfikowana przez funkcję zad3standard

3.4 Wnioski

Dla $k \in Z$ niewykraczającego poza ustalony zakres eksponenty:
1. Liczby w przedziale $[2^{k-1}, 2^k)$ są równomiernie rozmieszczone.
2. Dokładność obliczeń może zależeć od przedziału $[2^{k-1}, 2^k)$, do którego wpada dana liczba zmiennoprzecinkowa.
3. Jeśli w przedziale $[2^{k-1}, 2^k)$ krok wynosi δ, to w przedziale $[2^k, 2^{k+1})$ wynosi ona 2δ. Ten fakt uświadamia nam, że wraz z każdym takim przedziałem krok wzrasta dwukrotnie, a co za tym idzie prezycja arytmetyki maleje.

4 Zadanie 4

4.1 Opis zadania

Zadanie polegało na wyznaczeniu liczby $x \in (1, 2)$ takiej, że $x \cdot \frac{1}{x} \neq 1$ wykonując obliczenia w arytmetyce double (Float64 w języku Julia). Taką liczbę należało najpierw wyznaczyć eksperymentalnie, a następnie wyznaczyć najmniejszą taką liczbę.

4.1.1 Wyznaczanie eksperymentalne - opis rozwiązania

Do znalezienia zadanej liczby użyto algorytmu:

```
function zad4exp()
    z = Float64(1.01)
    while(z*(1.0/z))==1.0
        z = prevfloat(z)
    end
    return z
end
```

Takich liczb można się spodziewać przede wszystkim na krańcach przedziału $(1, 2)$. Dlatego szukamy po wszystkich liczbach maszynowych z zakresu $(1, 1.1]$ rozpoczynając obliczenia od liczby 1.1, a następnie sprawdzamy warunek na sąsiedniej mniejszej liczbie maszynowej. Obliczenia wykonujemy dopóki zachowana jest równość warunkująca wykonywanie pętli while.

Wywołanie tej funkcji zwraca wynik 1.0099999999999634. Możemy dodatkowo sprawdzić poprawność otrzymanego wyniku za pomocą interpretera języka Julia:

```
Float64(1.0099999999999634*((1/(1.0099999999999634)))) = 0.9999999999999999
```

Otrzymany przez nas wynik jest poprawny - dla otrzymanej wartości zmiennej x równość nie jest zachowana.

4.1.2 Wyznaczanie najmniejszej liczby $x > 1$ niespełniającej równania

Do znalezienia tej liczby używamy algorytmu:

```
function zad4smallest()
    z = Float64(1.0)
    while(z*(1.0/z))==1.0
        z = nextfloat(z)
    end
    println(z)
    return z
end
```

Sposób działania algorytmu jest następujący: przeszukujemy wszystkie liczby maszynowe rozpoczynając od jedynki i rozważając kolejne sąsiadujące z prawej strony liczby maszynowe.

Wywołanie funkcji zwraca następujący wynik:

```
zad4smallest() = 1.000000057228997
```

4.1.3 Wnioski

Podczas obliczeń w arytmetyce float niektóre tożsamości algebraiczne mogą nie zostać zachowane. Dla specyficznych danych błędy wynikające z działań w danej arytmetyce mogą powodować zakłócenia w wynikach.

5 Zadanie 5 - obliczanie iloczynu skalarnego

5.1 Opis problemu

Zadanie polega na obliczeniu iloczynu skalarnego dwóch wektorów o długości $n = 5$ w arytmetykach single oraz double (Float32 i Float64) na kilka różnych sposobów: dodając do sumy mnożonych razy siebie liczb od lewej do prawej strony, od prawej do lewej strony, dodając otrzymane iloczyny rosnąco (iloczyny ujemne malejąco) oraz malejąco (iloczyny ujemne rosnąco).

5.2 Analiza wyników

W poniższej tabeli przedstawiono wartości zaimplementowanych funkcji dla $x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$ i $y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$. Funkcje użyte do obliczenia wartości iloczynów skalarnych znajdują się w pliku zad5.jl

sposób	arytmetyka Float32	arytmetyka Float64
dodawanie iloczynów do sumy od lewej do prawej strony	-0.4999443	1.0251881368296672e-10
dodawanie iloczynów do sumy od prawej do lewej strony	-0.4543457	-1.5643308870494366e-10
dodawanie dodatnich iloczynów malejąco, a ujemnych rosnąco	-0.25	0.0
dodawanie dodatnich iloczynów rosnąco, a ujemnych malejąco	-0.25	0.0

Tabela 6: Porównanie wartości iloczynów skalarnych wyznaczonych na różne sposoby dla arytmetyk Float32 i Float64

5.3 Wnioski

Kolejnośc wykonywania operacji w arytmetyce zmiennopozycyjnej może mieć znaczenie, a tego dowodem są różne wartości sum wyznaczone przez zaimplementowane algorytmy.

6 Zadanie 6

6.1 Opis zadania

Zadanie polega na wyznaczeniu wartości funkcji $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla $x = 8^{-1}, 8^{-2}, 8^{-3}...$ i określeniu, które wyniki są wiarygodne a które nie.

6.2 Opis rozwiązania

Poniżej przedstawiono funkcje użyte do obliczenia wartości funkcji f i g:

```
function f(x)
    return Float64(sqrt((x*x)+1)-1)
end

function g(x)
    return Float64((x*x)/(sqrt((x*x)+1)+1))
end

function zad6(k)
    y = Float64(0.125)
    i = Int(1)
    while i<k
        y = (0.125 * y)
        i = i + 1;
    end
end
```

Funkcja x jest źle uwarunkowana. Dla bardzo małych wartości x mamy do czynienia z odejmowaniem dwóch liczb bardzo bliskich sobie, co może powodować, że dla pewnych małych wartości x otrzymamy $f(x) = 0$. Przekształcenie tej funkcji do postaci, w której unikniemy odejmowania liczb bliskich sobie może zwiększyć precyzję obliczeń. Poniżej przedstawiono wartości funkcji f i g wyliczone w arytmetyce Float64:

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
8^{-4}	2.9802321943606103e-8	2.9802321943606116e-8
8^{-5}	4.656612873077393e-10	4.6566128719931904e-10
8^{-6}	7.275957614183426e-12	7.275957614156956e-12
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
8^{-11}	0.0	6.776263578034403e-21
8^{-12}	0.0	1.0587911840678754e-22
8^{-13}	0.0	1.6543612251060553e-24
8^{-14}	0.0	2.5849394142282115e-26
8^{-15}	0.0	4.0389678347315804e-28
8^{-16}	0.0	6.310887241768095e-30
8^{-17}	0.0	9.860761315262648e-32
8^{-18}	0.0	1.5407439555097887e-33
8^{-19}	0.0	2.407412430484045e-35
8^{-20}	0.0	3.76158192263132e-37
8^{-21}	0.0	5.877471754111438e-39
8^{-22}	0.0	9.183549615799121e-41
8^{-23}	0.0	1.4349296274686127e-42
8^{-24}	0.0	2.2420775429197073e-44
8^{-25}	0.0	3.503246160812043e-46
8^{-26}	0.0	5.473822126268817e-48
8^{-27}	0.0	8.552847072295026e-50
8^{-28}	0.0	1.3363823550460978e-51
8^{-29}	0.0	2.088097429759528e-53
8^{-30}	0.0	3.2626522339992623e-55
...	...	...
8^{-176}	0.0	6.4758e-319
8^{-177}	0.0	1.012e-320
8^{-178}	0.0	1.6e-322
8^{-179}	0.0	0.0

Tabela 7: Porównanie wartości funkcji f z wartościami funkcji g dla kolejnych ujemnych potęg liczby 8

Już dla $x = 8^{-9}$ wartość funkcji f wynosi 0 i wartość ta nie zmienia się dla kolejnych argumentów. Funkcja g pozwala na wyliczenie wartości wyrażenia bez anomalii związanych ze złym uwarunkowaniem funkcji.

6.3 Wnioski

1. Wartości funkcji f dla małych wartości x są niewiarygodne ze względu na złe uwarunkowanie tej funkcji. Chcąc skorzystać z wartości tego wyrażenia należy użyć funkcji g.

2. Źle uwarunkowana funkcja potrafi zwracać mocno zaburzone wyniki

3. Jeśli to możliwe, źle uwarunkowaną funkcję należy przekształcić do takiej postaci, aby wykonywanie niebezpiecznych operacji się nie odbywało.

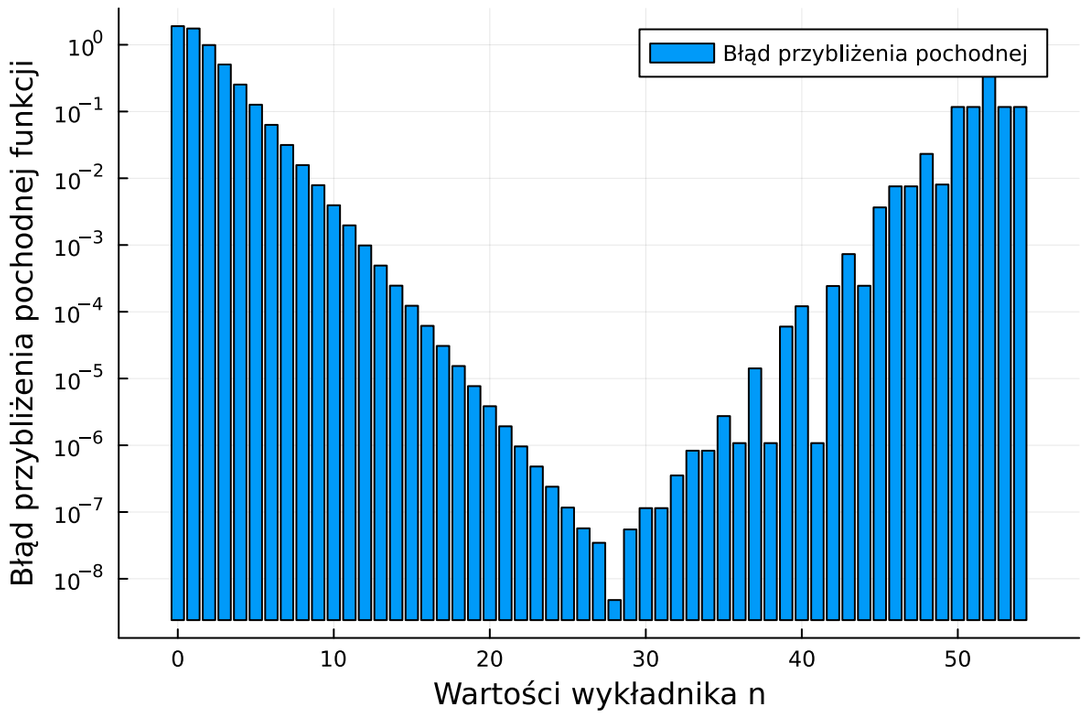
7 Zadanie 7

7.1 Opis zadania

Zadanie polegało na obliczeniu oraz analizie błędów obliczeń pochodnej funkcji $f(x) = sin(x) + cos(3x)$ za pomocą wzoru $f'(x) \approx \widetilde{f}'(x_0) = \frac{f(x_0+h)-f(x_0)}{h}$ przyjmując $x_0 = 1$ oraz na przeanalizowaniu zmian wartości pochodnej oraz błędu dla $h = 2^0$, 2^1, 2^2, ... 2^{54}. Obliczenia mają być przeprowadzone w arytmetyce double (Float64)

7.2 Opis rozwiązania

Pochodną funkcji f można obliczyć ze wzoru $f'(x) = cos(x) - 3sin(3x)$. Wartość df obliczona w punkcie $x_0 = 1$ dla arytmetyki Float64 wynosi 0.11694228168853815. Błąd wyraża się wzorem $|\widetilde{f}'(x_0) - f'(x_0)|$. Dla zadanych wartości h obliczamy wartość przybliżonej pochodnej oraz błąd. Wyniki zostały wygenerowane za pomocą funkcji zad7() znajdującej się w pliku zad7.jl.



Wykres 1: Błąd przybliżenia pochodnej dla kolejnych wykładników $h = 2^0$, 2^1, 2^2, ... 2^{54}. Na osi OY zastosowano skalę logarytmiczną.

h	$\tilde{f}'(x_0)$	$\lvert\tilde{f}'(x_0) - f'(x_0)\rvert$	$1 + h$
2^{0}	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	1.8704413979316472	1.753499116243109	1.5
2^{-2}	1.1077870952342974	0.9908448135457593	1.25
2^{-3}	0.6232412792975817	0.5062989976090435	1.125
2^{-4}	0.3704000662035192	0.253457784514981	1.0625
2^{-5}	0.24344307439754687	0.1265007927090087	1.03125
2^{-6}	0.18009756330732785	0.0631552816187897	1.015625
2^{-7}	0.1484913953710958	0.03154911368255764	1.0078125
2^{-8}	0.1327091142805159	0.015766832591977753	1.00390625
2^{-9}	0.1248236929407085	0.007881411252170345	1.001953125
2^{-10}	0.12088247681106168	0.0039401951225235265	1.0009765625
2^{-11}	0.11891225046883847	0.001969968780300313	1.00048828125
2^{-12}	0.11792723373901026	0.0009849520504721099	1.000244140625
2^{-13}	0.11743474961076572	0.0004924679222275685	1.0001220703125
2^{-14}	0.11718851362093119	0.0002462319323930373	1.00006103515625
2^{-15}	0.11706539714577957	0.00012311545724141837	1.000030517578125
2^{-16}	0.11700383928837255	6.155759983439424e-5	1.0000152587890625
2^{-17}	0.11697306045971345	3.077877117529937e-5	1.0000076293945312
2^{-18}	0.11695767106721178	1.5389378673624776e-5	1.0000038146972656
2^{-19}	0.11694997636368498	7.694675146829866e-6	1.0000019073486328
2^{-20}	0.11694612901192158	3.8473233834324105e-6	1.0000009536743164
2^{-21}	0.1169442052487284	1.9235601902423127e-6	1.0000004768371582
2^{-22}	0.11694324295967817	9.612711400208696e-7	1.000000238418579
2^{-23}	0.11694276239722967	4.807086915192826e-7	1.0000001192092896
2^{-24}	0.11694252118468285	2.394961446938737e-7	1.0000000596046448
2^{-25}	0.116942398250103	1.1656156484463054e-7	1.0000000298023224
2^{-26}	0.11694233864545822	5.6956920069239914e-8	1.0000000149011612
2^{-27}	0.11694231629371643	3.460517827846843e-8	1.0000000074505806
2^{-28}	0.11694228649139404	4.802855890773117e-9	1.0000000037252903
2^{-29}	0.11694222688674927	5.480178888461751e-8	1.0000000018626451
2^{-30}	0.11694216728210449	1.1440643366000813e-7	1.0000000009313226
2^{-31}	0.11694216728210449	1.1440643366000813e-7	1.0000000004656613
2^{-32}	0.11694192886352539	3.5282501276157063e-7	1.0000000002328306
2^{-33}	0.11694145202636719	8.296621709646956e-7	1.0000000001164153
2^{-34}	0.11694145202636719	8.296621709646956e-7	1.0000000000582077
2^{-35}	0.11693954467773438	2.7370108037771956e-6	1.0000000000291038
2^{-36}	0.116943359375	1.0776864618478044e-6	1.000000000014552
2^{-37}	0.11692810058593 75	1.4181102600652196e-5	1.000000000007276
2^{-38}	0.116943359375	1.0776864618478044e-6	1.000000000003638
2^{-39}	0.11688232421875	5.9957469788152196e-5	1.000000000001819
2^{-40}	0.1168212890625	0.0001209926260381522	1.0000000000009095
2^{-41}	0.116943359375	1.0776864618478044e-6	1.0000000000004547
2^{-42}	0.11669921875	0.0002430629385381522	1.0000000000002274
2^{-43}	0.1162109375	0.0007313441885381522	1.0000000000001137
2^{-44}	0.1171875	0.0002452183114618478	1.0000000000000568
2^{-45}	0.11328125	0.0036610316885381 52	1.0000000000000284
2^{-46}	0.109375	0.0075672816885381 52	1.0000000000000142
2^{-47}	0.109375	0.0075672816885381 52	1.000000000000007
2^{-48}	0.09375	0.0231922816885381 52	1.0000000000000036
2^{-49}	0.125	0.0080571831146184 78	1.0000000000000018
2^{-50}	0.0	0.1169422816885381 5	1.0000000000000009
2^{-51}	0.0	0.1169422816885381 5	1.0000000000000004
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.1169422816885381 5	1.0
2^{-54}	0.0	0.1169422816885381 5	1.0

Tabela 8: Wartość pochodnej funkcji f obliczanej za pomocą przybliżenia wraz z wartościami błędu oraz wartościami wyrażenia $1 + h$ dla $h = 2^0, 2^1, 2^2...$

7.3 Analiza wyników

Wartości błędu uzyskane dla początkowych wartości $h = 2^n$ nie są zaskakujące - są coraz mniejsze. Fakt ten wynika z definicji pochodnej funkcji w punkcie x_0 ($f'(x_0) = lim_{x \to 0} \frac{f(x_0+h)-f(x_0)}{h}$). Najmniejszy błąd uzyskano dla $h = 2^{-28}$ i wyniósł 4.802855890773117e-9. Dla kolejnych, coraz mniejszych wartości h, możemy zaobserwować tendencję wzostową błędu. Błąd ten nie ma uzasadnienia we własnościach pochodnej funkcji, lecz w specyficznym działaniu wykonywanym podczas obliczania tych wartości - obliczaniu wartości wyrażenia $f(x_0 + h) = f(1 + h)$. Gdy h jest bardzo małe przy obliczaniu wartości wyrażenia $1 + h$ w arytmetyce zmiennopozycyjnej mogą wystąpić anomalie związane z utratą bitów znaczących.

7.4 Wnioski

Tożsamości algebraiczne oraz formuły matematyczne muszą być poddane szczegółowej analizie, gdy w arytmetyce double chcemy z dużą dokładnością badać ich własności.