OBLICZENIA NAUKOWE

Lista nr 1.

Patrycja Paradowska
nr indeksu: 244952
Prowadzący laboratoria: Mgr inż. Marta Słowik

19.10.2019r.

Poniższe sprawozdanie obejmuje polecenia laboratoryjne z listy 1., mające na celu przybliżenie arytmetyki zmiennopozycyjnej (**IEEE 754**), lepsze zrozumienie i poznanie jej właściwości oraz zagrożeń związanych z wykonywaniem w niej obliczeń.

Spis treści

1 Zadanie 1. - Rozpoznanie arytmetyki

W zadaniu nr 1. należało wyznaczyć niektóre wartości istotne dla arytmetyk `Float16`, `Float32` i `Float64`, a mianowicie: epsilon maszynowy, wartość *eta*, ale także maksymalną wartość możliwą do uzyskania w wymienionych typach zmiennopozycyjnych. Językiem programowania, który został użyty do tych celów, była Julia.

1.1 Epsilon maszynowy

Epsilonem maszynowym *macheps* (ang. machine epsilon) nazywamy taką najmniejszą liczbę $macheps > 0$, że spełniona jest nierówność:

$$fl(1.0 + macheps) > 1.0$$

gdzie fl oznacza arytmetykę, w której wykonujemy powyższe działanie.

1.1.1 Przedstawienie problemu

Należało wyznaczyć iteracyjnie epsilony maszynowe dla wszystkich dostępnych typów zmiennopozycyjnych, zgodnych ze standardem **IEEE 754 (half, single, double).** . Ponadto wykonanie zadania obejmowało porównanie ich z wartościami zwracanymi przez funkcje `eps(Float16)`, `eps(Float32)`, `eps(Float64)` z języka Julia oraz z danymi znajdującymi się w pliku nagłówkowym `float.h` dowolnej instalacji języka C.

1.1.2 Algorytm

Aby wyznaczyć epsilon maszynowy, użyto następującego algorytmu:

```
1: function MACHEPS-CALCULATION
2:     macheps ← 1.0
3:     while 1.0 + macheps/2.0 > 1.0 do
4:         macheps ← macheps/2.0
5:     end while
6:     return macheps
7: end function
```

Polega on na dzieleniu w pętli while zmiennej macheps (zainicjalizowanej jedynką) przez dwa tak długo, aż zostanie ona potraktowana jako zero, to znaczy że: $1.0 \oplus macheps = 1.0$. Po zakończeniu wykonywania pętli, zmienna macheps będzie przechowywała ostatnią przypisaną wartość większą od zera, czyli poszukiwany epsilon maszynowy.

1.1.3 Uzyskane wyniki

Następnie otrzymane wyniki należało porównać z wartościami, które są zwracane przez funkcję eps() dostępną dla poszczególnych typów zmiennopozycyjnych oraz z danymi zawartymi w pliku nagłówkowym `float.h`. Uzyskane wyniki umieszczono w poniższej tabeli:

typ	macheps	funkcja eps()	float.h
Float16	0.000977	0.000977	—
Float32	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$
Float64	$2.220446049250313 \cdot 10^{-16}$	$2.220446049250313 \cdot 10^{-16}$	$2.220446049250313 \cdot 10^{-16}$

Tabela 1: Zestawienie obliczonego macheps dla różnych typów wraz z poprawnymi danymi oraz wartościami dla języka C z pliku float.h.

1.1.4 Obserwacje i wnioski

Uzyskane rozwiązania są identyczne, zatem można stwierdzić, że metoda iteracyjna obliczania epsilonu maszynowego jest poprawna. Wyniki uzyskane za jej pomocą pokrywają się z prawidłowymi wartościami. Pokazują, że im większa jest precyzja arytmetyki, tym mniejszy jest epsilon maszynowy. Wynika to z faktu, że w przypadku arytmetyki o większej precyzji możemy zapisać więcej cyfr znaczących, przez to rzadziej następuje zjawisko "ucinania bitów" i zaokrąglania. Epsilon maszynowy jest ściśle związany z precyzją arytmetyki zadaną wzorem 2^{-t-1}, gdzie t jest długością mantysy, jego wartość jest dwa razy większa (2^{-t}). Epsilon maszynowy określa najmniejszą wartość, która po dodaniu do dowolnej większej liczby wpływa na obliczenia.

1.2 ETA

1.2.1 Przedstawienie problemu

Kolejnym poleceniem było iteracyjne wyznaczenie liczby *eta* takiej, że:

$$eta > 0.0$$

dla wszystkich typów zmiennopozycyjnych `Float16`, `Float32` i `Float64` i porównanie jej z wartościami zwracanymi przez funkcje `nextfloat(Float16(0.0))`, `nextfloat(Float32(0.0))` i `nextfloat(Float64(0.0))` z języka Julia, a także liczbą MIN_{sub}.

1.2.2 Algorytm

W celu wyznaczenia liczby *eta* zaimplementowany został następujący algorytm:

```
1: function ETA-CALCULATION
2:     eta ← 1.0
3:     while eta/2.0 ≠ 0.0 do
4:         eta ← eta/2.0
5:     end while
6:     return eta
7: end function
```

Polega on na dzieleniu w pętli while wartości zmiennej eta (zainicjalizowanej jedynką) przez dwa tak długo, aż zostanie ona potraktowana jako zero (tzn. eta = 0.0). Po zakończeniu wykonywania pętli w zmiennej eta pozostanie poszukiwana wartość.

1.2.3 Uzyskane wyniki

W tabeli poniżej przedstawiono otrzymane wartości liczbowe zwrócone przez iteracyjny algorytm wraz z wynikem działania funkcji nextfloat() dla odpowiednich typów zmiennopozycyjnych:

typ	**eta**	**nextfloat(typ)**
Float16	$6.0 \cdot 10^{-8}$	$6.0 \cdot 10^{-8}$
Float32	$1.0 \cdot 10^{-45}$	$1.0 \cdot 10^{-45}$
Float64	$5.0 \cdot 10^{-324}$	$5.0 \cdot 10^{-324}$

Tabela 2: Zestawienie obliczonego *eta* dla różnych typów wraz z poprawnymi wartościami.

1.2.4 Obserwacje i wnioski

Wartości eta obliczone w metodzie iteracyjnej są takie same jak zwracane przez funkcję nextfloat. Dzięki temu można wnioskować, że przedstawiony algorytm jest poprawny. Wraz ze wzrostem precyzji arytmetyki maleje wartość eta. Zmniejszanie się wartości eta odwrotnie proporcjonalne do precyzji arytmetyki wynika z tego, że z powodu zwiększonej długości mantysy, można reprezentować mniejsze liczby. Liczba *eta* ma ścisły związek z liczbą MIN_{sub}. Jest najmniejszą możliwą do zapisania liczbą dodatnią. Pokazuje to jej zapis bitowy. W arytmetyce `Float64`:

"0000000000000000000000000000000000000000000000000000000000000001"

A w arytmetyce `Float32`:

"00000000000000000000000000000001"

Zadziwiający może być fakt, że w dowolnej arytmetyce wszystkie bity cechy to 0. Oznacza to zatem, że jest to liczba zdenormalizowana (*subnormal*). W liczbie *eta* ostatni bit mantysy wynosi 1, a więc otrzymana *eta* jest faktycznie najmniejszą liczbą dodatnią, jaką da sie zapisać w tej arytmetyce. Reasumując $eta = MIN_{sub}$. Inne funkcje, jak chociażby floatmin, zwracają wartości znormalizowane, czyli $floatmin = MIN_{nor}$.

1.3 MAX

1.3.1 Przedstawienie problemu

Celem trzeciego podpunktu z zadania 1. było iteracyjne wyznaczenie przy użyciu języka Julia maksymalnej wartości możliwej do zapisania (liczby MAX) dla wszystkich typów zmiennopozycyjnych `Float16`, `Float32` i `Float64`,zgodnych ze standardem **IEEE 754 (half, single, double).**, a następnie porównanie jej z wartościami zwracanymi przez funkcję *floatmax* oraz z danymi zawartymi w pliku nagłówkowym `float.h` języka C.

1.3.2 Algorytm

Aby wyznaczyć liczbę `max` $< \infty$ w iteracyjny sposób, został stworzony prosty program w języku Julia, który realizuje poniższy algorytm:

```
1: function MAX-CALCULATION
2:     max ← 1.0
3:     while max · 2.0 ≠ ∞ do
4:         max ← 2.0 · max
5:     end while
6:     max ← (2.0 − macheps) · max
7:     return max
8: end function
```

Polega on na mnożeniu w pętli wartości zmiennej `max` (zainicjalizowanej jedynką) przez dwa, tak długo, dopóki nie przyjmie ona wartości ∞. Skorzystano tutaj z funkcji *isinf()*, która zwraca wartość true, jeśli argument jest nieskończonością. W ostatnim kroku jest wykonywane mnożenie przez liczbę $2.0 - $`macheps`(), co pozwala uzyskać największą skończoną liczbę dostępną w danym typie.

1.3.3 Uzyskane wyniki

Porównanie wyników zwróconych przez przedstawiony algorytm z wartościami funkcji *floatmax* oraz z zamieszczonymi w pliku nagłówkowym *float.h* przedstawiono w poniższej tabeli:

typ	MAX iteracyjnie	floatmax(typ)	float.h
Float16	$6.55 \cdot 10^4$	$6.55 \cdot 10^4$	—
Float32	$3.4028235 \cdot 10^{38}$	$3.4028235 \cdot 10^{38}$	$3.4028235 \cdot 10^{38}$
Float64	$1.7976931348623157 \cdot 10^{308}$	$1.7976931348623157 \cdot 10^{308}$	$1.7976931348623157 \cdot 10^{308}$

Tabela 3: Zestawienie obliczonego *MAX* z poprawnymi wartościami.

1.3.4 Obserwacje i wnioski

Uzyskano rozwiązania identyczne ze zwracanymi przez funkcję *floatmax* oraz z maksymalnymi wartościami deklarowanymi w dokumentacji języka C, co dowodzi, że metoda iteracyjna obliczania liczby *MAX* jest poprawna. Na podstawie przedstawionych w tabeli wartości można wyciągnąć wniosek, że wraz ze wzrostem precyzji arytmetyki zwiększa się również wyliczona wartość *MAX*. Wynika to z faktu, iż przy większej precyzji możemy zapisać w danej arytmetyce więcej liczb.

2 Zadanie 2. - Epsilon maszynowy Kahana

William Kahan to informatyk i matematyk specjalizujący się w metodach numerycznych. Stwierdził on, że epsilon maszynowy można uzyskać obliczając wartość wyrażenia:

$$3(4/3 - 1) - 1$$

w arytmetyce zmiennopozycyjnej.

2.1 Przedstawienie problemu

Zadanie polega na eksperymentalnym sprawdzeniu w języku Julia słuszności twierdzenia Kahana dla wszystkich typów zmiennopozycyjnych `Float16`, `Float32` i `Float64`.

2.2 Algorytm

Obliczono wartość wyrażenia podanego przez Kahana dla wszystkich typów zmiennopozycyjnych wykorzystując odpowiednie rzutowania typów:

```
FloatX(3.0) * (FloatX(4.0)/FloatX(3.0) - FloatX(1.0)) - FloatX(1.0)
```

gdzie za `X` wstawiono kolejno `16`, `32` i `64`.

2.3 Uzyskane wyniki

Otrzymane wyniki dla poszczególnych typów zmiennopozycyjnych wypisano na ekran i powrównano w tabeli z wartościami funkcji bibliotecznej `eps(FloatX)`:

Arytmetyka	Kahan	`eps()`
`Float16`	-0.000977	0.000977
`Float32`	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$
`Float64`	$-2.220446049250313 \cdot 10^{-16}$	$2.220446049250313 \cdot 10^{-16}$

Tabela 4: Zestawienie obliczonego *macheps* Kahana dla różnych typów wraz z poprawnymi wartościami.

2.4 Obserwacje i wnioski

Powyższa tabela pokazuje, że prawidłowe rozwiązanie udało się uzyskać jedynie dla typu `Float32`. Dla typów `Float16` i `Float64` wyniki różnią się jednak co do znaku. Stwierdzenie Kahana byłoby słuszne, gdyby z wartości wyrażenia wziąć jego wartość bezwzględną. Jednak jego rozumowanie było w gruncie rzeczy poprawne i korzystał on z niemożliwości dokładnego przedstawienia liczby 4/3 w systemie dwójkowym. Celowo zastosował niedokładność zaokrąglenia do wyliczenia *epsilonu maszynowego*. W 2 przypadkach ujemność wyników jest skutkiem parzystości mantysy typów i faktu, że w tym wypadku w rozwinięciu dwójkowym liczby 4/3 na ostatniej pozycji mantysy występuje 0. Zgodnie z zasadą *"round to even"* liczba zaokrąglana jest z niedomiarem i to przy dalszych obliczeniach powoduje wynik ujemny.

3 Zadanie 3. - Badanie rozmieszczenia liczb zmiennopozycyjnych w arytmetyce Float64

3.1 Przedstawienie problemu

W zadaniu należało napisać w języku Julia program, który eksperymentalnie sprawdzi, że w arytmetyce `Float64` liczby zmiennopozycyjne są równomiernie rozmieszczone w przedziale $[1, 2]$ z krokiem $\delta = 2^{-52}$. Równoznaczne jest to ze stwierdzeniem, że każda liczby zmiennopozycyjna x z zakresu $[1, 2]$ może zostać przedstawiona jako $x = 1 + k \cdot \delta$ w danej arytmetyce, gdzie $k \in \{1, 2, \ldots, 2^{52} - 1\}$, a $\delta = 2^{-52}$. Należało także sprawdzić, jak rozmieszczone są liczby zmiennopozycyjne w przedziałach $[\frac{1}{2}, 1]$ oraz $[2, 4]$ i jak mogą być przedstawione dla rozpatrywanego przedziału.

3.2 Algorytm

W celu rozwiązania zadania użyto algorytmu wyświetlającego liczby z zadanego przedziału powiększane o wskazaną wartość δ w zadanej liczbie kroków k ($x = 1 + k\delta$, gdzie $k = 1, 2, \ldots$). Wypisano je na ekran w postaci bitowej (funkcja `bitstring`(x)). To samo postępowanie powtórzono dla przedziałów $[\frac{1}{2}, 1]$ oraz $[2, 4]$, przy czym przyjęto hipotezę, że $\delta_{[\frac{1}{2},1]} = 2^{-53}$ oraz $\delta_{[2,4]} = 2^{-51}$.

```
1: liczba ← start
2: for i ← 1 to iteracje do
3:     liczba ← liczba + δ
4:     print bitstring(liczba)
5: end for
```

3.3 Uzyskane wyniki

W poniższej tabeli przedstawiono rezultaty działania programu dla odpowiednio wybranych wartości δ, 2^{-52} dla $[1, 2]$, 2^{-53} dla $[\frac{1}{2}, 1]$ i 2^{-51} dla $[2, 4]$. Wybranych zostało kilka pierwszych i ostatnich liczb dla każdego przedziału. Z prezentacji bitowej wynika, że są to kolejne liczby w arytmetyce `Float64`, ponieważ mają one tę samą cechę, a mantysa każdej kolejnej liczby (powiększonej o δ) jest większa o 1.

Przedział $[1, 2]$		**Przedział $[\frac{1}{2}, 1]$**		**Przedział $[2, 4]$**	
x	`bitstring`(x)	x	`bitstring`(x)	x	`bitstring`(x)
1.0	$0011111111110\ldots000$	0.5	$0011111111100\ldots000$	2.0	$0100000000000\ldots000$
$1.0+\delta$	$0011111111110\ldots001$	$0.5+\delta$	$0011111111100\ldots001$	$2.0+\delta$	$0100000000000\ldots001$
$1.0+2\delta$	$0011111111110\ldots010$	$0.5+2\delta$	$0011111111100\ldots010$	$2.0+2\delta$	$0100000000000\ldots010$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2.0-2\delta$	$0011111111111\ldots110$	$1.0-2\delta$	$0011111111101\ldots110$	$4.0-2\delta$	$1000000000011\ldots110$
$2.0-\delta$	$0011111111111\ldots111$	$1.0-\delta$	$0011111111101\ldots111$	$4.0-\delta$	$1000000000011\ldots111$
2.0	$0100000000000\ldots000$	1.0	$0011111111110\ldots000$	4.0	$1000000000100\ldots000$

Tabela 5: Kolejne liczby z przedziałów $[1, 2]$, $[\frac{1}{2}, 1]$, $[2, 4]$ w zapisie bitowym. Przyjęto $\delta_{[1,2]} = 2^{-52}$, $\delta_{[\frac{1}{2},1]} = 2^{-53}$ oraz $\delta_{[2,4]} = 2^{-51}$.

3.4 Obserwacje i wnioski

Z analizy danych w tabeli wynika, że między kolejnymi potęgami dwójki liczby zmiennopozycyjne są równomiernie rozmieszczone. Patrząc na prezentację bitową można łatwo zauważyć, że liczby między kolejnymi potęgami dwójki posiadają tę samą cechę, a zmianie ulega tylko mantysa. Liczb pomiędzy kolejnymi potęgami dwójki jest więc tyle samo, a co za tym idzie wraz ze zwiększaniem się przedziału maleje ich gęstość.

4 Zadanie 4. - Problem nieodwracalności dzielenia

4.1 Przedstawienie problemu

Celem ćwiczenia było eksperymentalne wyznaczenie w arytmetyce `Float64` liczby zmiennopozycyjnej x z przedziału $(1, 2)$, takiej że:

$$x \cdot \frac{1}{x} \neq 1, \tag{1}$$

tzn. obliczyć wyrażenie $fl(x \cdot fl(\frac{1}{x})) \neq 1$, gdzie fl oznacza arytmetykę, w której chcemy wykonać powyższe działanie. Należało znaleźć najmniejszą liczbę spełniającą powyższą zależność.

4.2 Algorytm

```
1: function SMALLST
2:     x ← NEXTFLOAT(1.0)
3:     while x · 1.0/x = 1.0 and x < 2.0 do
4:         x ← NEXTFLOAT(x)
5:     end while
6:     return x
7: end function
```

Wyrażenie (1) ma dużą liczbę rozwiązań i znalezienie ich wszystkich trwałoby wiele czasu, zatem powyższa procedura zwraca jedynie najmniejszą liczbę spełniającą zadaną zależność (znajdującą się w przedziale (1,2). W celu rozwiązania zadania dla kolejnych liczb x w arytmetyce `Float64`, zaczynając od najmniejszej liczby większej od 1, zostało sprawdzone czy warunek $x \cdot (1/x) \neq 1$ zachodzi. W momencie znalezienia pierwszej takiej liczby program przerywał pracę i wyświetlał wynik na ekranie. Jeśli chcielibyśmy wyznaczyć wartość największą z tego przedziału, która spełniałaby nasze założenia, to musielibyśmy zmienną x zainicjalizować wartością 2.0, a w pętli zamiast funkcji *nextfloat()*, użyć *prevfloat()*.

4.3 Uzyskane wyniki

Najmniejszą eksperymentalnie wyznaczoną liczbą, spełniającą podane warunki zadania jest:

$$1.000000057228997$$

4.4 Obserwacje i wnioski

Zadanie pokazuje, że działania arytmetyczne na liczbach zmiennopozycyjnych mogą generować błędy związane z zaokrąglaniem wyliczonych wartości. Przy używaniu typów zmiennopozycyjnych takie błędy często są nieuniknione. Mimo, że matematycznie równanie x * 1/x = 1 spełnia każda niezerowa liczba (tzn. operacja dzielenia jest odwracalna), w arytmetyce zmiennoprzecinkowej wcale nie musi tak być, z powodu ograniczonej liczby bitów przeznaczonych do zapisu liczby i związanych z tym błędów zaokrągleń.

5 Zadanie 5. - Iloczyn skalarny

5.1 Przedstawienie problemu

W zadaniu 5. należało przeprowadzić eksperyment polegający na obliczeniu (w arytmetykach Float32 i Float64) iloczynu skalarnego zadanych dwóch wektorów:

$$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$
$$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$$

5.2 Algorytm

Należało zastosować cztery różne algorytmy, na różne sposoby obliczające iloczyn skalarny zgodnie ze specyfkacją zadania:

1. "w przód": $\sum_{i=1}^{n} x_i y_i$; czyli zaczynamy obliczanie iloczynu skalarnego od pierwszych współrzędnych

```
1: S ← 0
2: for i ← 1 to n do do
3:     S ← S + x_i · y_i
4: end for
```

2. "w tył": $\sum_{i=n}^{1} x_i y_i$; czyli zaczynamy obliczanie iloczynu skalarnego od ostatnich współrzędnych

```
1: S ← 0
2: for i ← 1 downto n do
3:     S ← S + x_i y_i
4: end for
```

3. dodanie dodatnich liczb w porządku od największej do najmniejszej oraz ujemnych w porządku od najmniejszej do największej, a następnie dodanie do siebie obliczonych sum częściowych; zostało to wykonane za pomocą sortowania i odpowiedniego dodania elementów tablicy sum częściowych;

4. od najmniejszego do największego - metoda przeciwna do sposobu *3.*

5.3 Uzyskane wyniki

Otrzymano następujące wyniki (Tabela 6):

typ	w przód	w tył	malejący	rosnący
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	$1.0251881368296672 \cdot 10^{-10}$	$-1.5643308870494366 \cdot 10^{-10}$	0.0	0.0

Tabela 6: Iloczyn skalarny danych wektorów.

Podana w zadaniu prawidłowa wartość iloczynu skalarnego z dokładnością do 15 cyfr to $-1.00657107000000 \cdot 10^{-11}$. Wszystkie otrzymane wyniki są od niej różne.

5.4 Obserwacje i wnioski

Zadanie pokazuje, że kolejność wykonywania działań nie jest bez znaczenia i może wywrzeć duży wpływ na otrzymywany wynik. Można zauważyć także, że im więcej wykonywanych jest działań na liczbach zmiennopozycyjnych tym większy jest błąd względny. Jednym ze sposobów na uniknięcie dużych błędów, kiedy inne metody zawodzą, jest użycie arytmetyki o większej precyzji. Użycie `Float64` zamiast `Float32` w zadaniu przybliżyło wynik do prawdy, jednak nadal jest to wartość odbiegająca od właściwej. Dane wektory są prawie ortogonalne, co powoduje, że próba obliczenia ich iloczynu skalarnego skutkuje popełnieniem dużych błędów względnych.

6 Zadanie 6. - Przybliżenie funkcji

6.1 Przedstawienie problemu

Zadanie 6. polegało na obliczeniu w języku Julia w arytmetyce `Float64` wartości funkcji:

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

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3} \dots$.
W poleceniu otrzymaliśmy informacje, że pomimo, iż f = g, to komputer daje różne wyniki. Należało roztrzygnąć, które są wiarygodne.

6.2 Algorytm

Dla kolejnych argumentów x w pętli zostały wykonane obliczenia wartości funkcji f i g.

6.3 Uzyskane wyniki

Wyniki działania programu dla 10 kolejnych wartości przedstawiono w Tabeli nr 7.

x	$f(x)$	$g(x)$
$x = 8^{-1}$	$f(x) = 0.0077822185373186414$	$g(x) = 0.0077822185373187065$
$x = 8^{-2}$	$f(x) = 0.00012206286282867573$	$g(x) = 0.00012206286282875901$
$x = 8^{-3}$	$f(x) = 1.9073468138230965 \cdot 10^{-6}$	$g(x) = 1.907346813826566 \cdot 10^{-6}$
...	...	...
$x = 8^{-8}$	$f(x) = 1.7763568394002505 \cdot 10^{-15}$	$g(x) = 1.7763568394002489 \cdot 10^{-15}$
$x = 8^{-9}$	$f(x) = 0.0$	$g(x) = 2.7755575615628914e \cdot 10^{-17}$
...	...	...
$x = 8^{-178}$	$f(x) = 0.0$	$g(x) = 1.6 \cdot 10^{-322}$
$x = 8^{-179}$	$f(x) = 0.0$	$g(x) = 0.0$
...	...	...
$x = 8^{-199}$	$f(x) = 0.0$	$g(x) = 0.0$
$x = 8^{-200}$	$f(x) = 0.0$	$g(x) = 0.0$

6.4 Obserwacje i wnioski

Funkcje f i g są sobie równe i dla kilku początkowych argumentów ich wartości są naprawde zbliżone. Jednak funkcja f bardzo szybko (dla 8^{-9}) zaczęła zwracać 0.0. Funkcja g jeszcze dla $x = 8^{-178}$ pokazuje wartość różną od zera ($1.6 \cdot 10^{-322}$).Pozwala to przypuszczać, że to właśnie jest bardziej wiarygodna funkcja. Dane w zadaniu funkcje dla $x \to 0$ dążą do zera, teoretycznie nigdy nie powinny tego zera osiągnąć. Arytmetyka komputera nie jest jednak na tyle dokładna żeby na to pozwolić. Funkcja f jednak osiąga wartość zero dla stosunkowo dużych x. Spowodowane jest to wykonywaniem odejmowania na wartościach do siebie zbliżonych. Funkcja g jest dużo bardziej dokładna, a jej błąd może wynikać z niedokładności arytmetyki. Aby unikać ryzykownego odejmowania od siebie bliskich liczb, można zastosować wyrażenie w innej alternatywnej postaci lub użyć większej precyzji.

7 Zadanie 7. - Przybliżenie pochodnej

7.1 Przedstawienie problemu

Nasz problem w ostatnim zadaniu polega na policzeniu w arytmetyce `Float64` przybliżonej wartości pochodnej funkcji:

$$f(x) = \sin x + \cos 3x$$

w punkcie $x_0 = 1$ oraz błędów $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$ $(n = 0, 1, 2, \ldots, 54)$.

7.2 Algorytm

Żeby obliczyć przybliżoną wartość pochodnej funkcji zastosowano znany wzór:

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

W prosty sposób wyprowadzono także wzór na pochodną funkcji:

$$f'(x) = \cos(x) - 3 \cdot \sin(3x)$$

aby dokonać obliczenia błędu. Dla kolejnych wartości h w pętli została obliczona przybliżona pochodna, błąd bezwzględny oraz wartość $h + 1$.

7.3 Uzyskane wyniki

Wyniki obliczeń dla poszczególnych wartości h umieszczono w poniższej tabeli.

h	$\tilde{f'}(x)$	$\mid f'(x) - \tilde{f'}(x) \mid$	$1+h$
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^{-15}	0.11706539714577957	0.00012311545724141837	1.000030517578125
2^{-16}	0.11700383928837255	$6.155759983439424 \cdot 10^{-5}$	1.0000152587890625
2^{-17}	0.11697306045971345	$3.077877117529937 \cdot 10^{-5}$	1.0000076293945312
...	...	...	...
2^{-27}	0.11694231629371643	$3.460517827846843 \cdot 10^{-8}$	1.0000000074505806
2^{-28}	0.11694228649139404	$4.802855890773117 \cdot 10^{-9}$	1.0000000037252903
2^{-29}	0.11694222688674927	$5.480178888461751 \cdot 10^{-8}$	1.0000000018626451
2^{-30}	0.11694216728210449	$1.1440643366000813 \cdot 10^{-7}$	1.0000000009313226
...	...	...	...
2^{-36}	0.116943359375	$1.0776864618478044 \cdot 10^{-6}$	1.000000000014552
2^{-37}	0.1169281005859375	$1.4181102600652196 \cdot 10^{-5}$	1.000000000007276
2^{-38}	0.116943359375	$1.0776864618478044 \cdot 10^{-6}$	1.000000000003638
...	...	...	...
2^{-51}	0.0	0.11694228168853815	1.0000000000000004
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.11694228168853815	1.0
2^{-54}	0.0	0.11694228168853815	1.0

7.4 Obserwacje i wnioski

Początkowo zmniejszanie wartości h przynosi oczekiwane skutki i błędy w obliczaniu przybliżonej pochodnej maleją przez pierwsze 28 iteracji, najdokładniejszy wynik uzyskano dla $h = 2^{-28}$, czyli n = 28. Następne zmniejszanie wartości h dokonywało coraz większej utraty dokładności obliczeń, błędy zaczęły z powrotem rosnąć. Wartości 1+h wraz z dalszym zmiejszaniem się wartości h są coraz mniej dokładne, a dla ostatnich h są równe 1.0.

Analiza wyników wyrażenia $1+h$ uzmysławia, że w pewnym momencie obliczania przybliżonej pochodnej h stało się na tyle małe w stosunku do 1, że 1 tak jakby "pochłonęło" h. Widać to dobrze przy ostatnich wartościach h, gdzie wynik operacji $1 + h = 1$. Taki wynik zaburza poprawność działania funkcji przybliżenia pochodnej. Dane wartości zwracają uwagę, że należy wystrzegać się dodawania do siebie liczb znacznie różniących się wykładnikami. Powoduje to błędy, mogące być nasilane przez dalsze obliczenia. Na zmniejszenie dokładności przybliżenia mogło mieć wpływ także odejmowanie bliskich sobie wartości $f(x_0 + h)$ i $f(x_0)$.