Obliczenia Naukowe
Lista 1

Jakub Gogola
236412

14 października 2018

1 Zadanie I

W zadaniu 1., używając języka Julia, należało wyznaczyć kilka wartości istotnych dla arytmetyk `Float16`, `Float32` i `Float64` - epsilon maszynowy, wartość *eta* oraz maksymalną wartość możliwą do uzyskania w wymienionych typach zmiennopozycyjnych.

1.1 Epsilon maszynowy

Epsilonem maszynowym (*macheps*) nazywamy taką najmniejszą liczbę $macheps > 0$, że $fl(1.0 + macheps) > 1.0$, gdzie fl oznacza arytmetykę, w której chcemy wykonać powyższe działanie.

1.1.1 Problem

W zadaniu należało, przy użyciu języka `Julia`, wyznaczyć metodą iteracyjną wartość *macheps* dla zadanych arytmetyk (`Float16`, `Float32`, `Float64`) oraz porównać otrzymane wyniki z wartościami zwracanymi przez funkcję `eps` oraz danymi zawartymi w pliku nagłówkowym `float.h` dla języka `C`.

1.1.2 Algorytm

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

Zakładamy, że wszelkie obliczenia zawarte w powyższym algorytmie są wykonywane w zadanej arytmetyce.

1.1.3 Wyniki

Dla poszczególnych typów arytmetyk zostały uzyskane następujące wyniki.

Arytmetyka `Float16`
Wartość uzyskana w wyniku działania algorytmu: 0.000977
Wartość domyślna dla arytmetyki `Float16`: 0.000977
Wartość dla języka `C` w pliku nagłówkowym `float.h`: *nie zdefiniowano*

Arytmetyka `Float32`
Wartość uzyskana w wyniku działania algorytmu: $1.1920929 \cdot 10^{-7}$

Wartość domyślna dla arytmetyki `Float32`: $1.1920929 \cdot 10^{-7}$
Wartość dla języka `C` w pliku nagłówkowym `float.h`: $1.1920929 \cdot 10^{-7}$

Arytmetyka `Float64`
Wartość uzyskana w wyniku działania algorytmu: $2.220446049250313 \cdot 10^{-16}$
Wartość domyślna dla arytmetyki `Float64`: $2.220446049250313 \cdot 10^{-16}$
Wartość dla języka `C` w pliku nagłówkowym `float.h`: $2.220446049250313 \cdot 10^{-16}$

1.1.4 Obserwacje

Wyniki otrzymane w wyniku działania algorytmu obliczającego wartość *macheps* są zgodne ze zwracanymi przez funkcję `eps`. Wraz ze wzrostem precyzji arytmetyki maleje wartość *macheps*.

1.1.5 Wnioski

Zastosowany algorytm do obliczania wartości *macheps* zwraca poprawne wyniki i jest to właściwy mechanizm obliczania tejże stałej. Zmniejszanie się wartości *macheps* wraz ze wzrostem precyzji arytmetyki wynika z faktu, że w przypadku arytmetyki o większej precyzji możemy zapisać więcej cyfr znaczących, przez to rzadziej występuje zjawisko "ucinania bitów" i zaokrąglania. Występuje ścisły związek liczby *macheps* z precyzją arytmetyki - określa ona najmniejszą wartość, która dodana do dowolnej, większej liczby wpływa jeszcze na obliczenia. Wartości *macheps* dla odpowiednich arytmetyk języka `C` zawarte w pliku `float.h` są takie same jak te dla języka `Julia`.

1.2 Eta

Liczbą *eta* nazywamy taką liczbę, że $eta > 0.0$ w zadanej arytmetyce, czyli jest to pierwsza liczba większa od wartości 0.0.

1.2.1 Problem

W zadaniu należało, przy użyciu języka `Julia`, wyznaczyć metodą iteracyjną liczbę *eta* dla zadanych arytmetyk (`Float16`, `Float32`, `Float64`) oraz porównać otrzymane wyniki z wartościami zwracanymi przez funkcję `nextfloat` oraz znaleźć związek otrzymanych wyników z wartością MIN_{sub}.

1.2.2 Algorytm

```
1: function COMPUTE-ETA
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
```

Zakładamy, że wszelkie obliczenia zawarte w powyższym algorytmie są wykonywane w zadanej arytmetyce.

1.2.3 Wyniki

W wyniku działania programu otrzymano następujące wyniki dla poszczególnych arytmetyk:

Arytmetyka `Float16`
Wartość uzyskana w wyniku działania algorytmu: $6.0 \cdot 10^{-8}$
Wartość domyślna dla arytmetyki `Float16`: $6.0 \cdot 10^{-8}$

Arytmetyka `Float32`
Wartość uzyskana w wyniku działania algorytmu: $1.0 \cdot 10^{-45}$
Wartość domyślna dla arytmetyki `Float32`: $1.0 \cdot 10^{-45}$

Arytmetyka `Float64`
Wartość uzyskana w wyniku działania algorytmu: $5.0 \cdot 10^{-324}$
Wartość domyślna dla arytmetyki `Float64`: $5.0 \cdot 10^{-324}$

1.2.4 Obserwacje

Wartości *eta* obliczone przez algorytm są takie same, jak te zwracane przez funkcję `nextfloat`. Wraz ze wzrostem precyzji arytmetyki maleje wartość *eta*.

1.2.5 Wnioski

Wnioskiem wynikającym z analizy otrzymanych wyników jest fakt, że zastosowany algorytm do obliczania wartości *eta* dla zadanych arytmetyk zwraca poprawny wynik zgodny ze stanem faktycznym. Zmniejszanie się wartości *eta* odwrotnie proporcjonalnie do precyzji arytmetyki wynika z faktu, że z powodu zwiększonej długości mantysy możemy reprezentować mniejsze liczby, stąd pierwsza możliwa do zapisania liczba większa od 0.0 jest mniejsza. Jest to najmniejsza możliwa do zapisania liczba większa od 0.0. Liczba *eta* ma ścisły związek z liczbą MIN_{sub}. Jej reprezentacja bitowa dla odpowiednich arytmetyk wygląda następująco:

Arytmetyka `Float32`: "00000000000000000000000000000001"
Arytmetyka `Float64`: "0000000000000000000000000000000000000000000000000000000000000001"

Jak widać, wszystkie bity cechy to 0. Jest to zatem liczba zdenormalizowana (*subnormal*).

1.3 MAX

Liczba *MAX* to największa możliwa do uzyskania wartość w zadanej arytmetyce zmiennopozycyjnej.

1.3.1 Problem

W zadaniu należało, przy użyciu języka `Julia`, wyznaczyć metodą iteracyjną wartość *MAX* dla zadanych arytmetyk (`Float16`, `Float32`, `Float64`) oraz porównać wyniki z wartościami zwracanymi przez funkcję `realmax` oraz danymi zwartymi w pliku nagłówkowym `float.h` dla języka `C`.

1.3.2 Algorytm

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

Zakładamy, że wszelkie obliczenia zawarte w powyższym algorytmie są wykonywane w zadanej arytmetyce.

1.3.3 Wyniki

W wyniku działania programu otrzymano następujące wyniki dla poszczególnych arytmetyk:

Arytmetyka `Float16`
Wartość uzyskana w wyniku działania algorytmu: $6.55 \cdot 10^4$
Wartość domyślna dla arytmetyki `Float16`: $6.55 \cdot 10^4$
Wartość dla języka `C` w pliku nagłówkowym `float.h`: *nie zdefiniowano*

Arytmetyka `Float32`
Wartość uzyskana w wyniku działania algorytmu: $3.4028235 \cdot 10^{38}$
Wartość domyślna dla arytmetyki `Float32`: $3.4028235 \cdot 10^{38}$
Wartość dla języka `C` w pliku nagłówkowym `float.h`: $3.4028235 \cdot 10^{38}$

Arytmetyka `Float64`
Wartość uzyskana w wyniku działania algorytmu: $1.7976931348623157 \cdot 10^{308}$
Wartość domyślna dla arytmetyki `Float64`: $1.7976931348623157 \cdot 10^{308}$
Wartość dla języka `C` w pliku nagłówkowym `float.h`: $1.7976931348623157 \cdot 10^{308}$

1.3.4 Obserwacje

Wartość MAX obliczone przez algorytm są takie same, jak te zwracane przez funkcję `realmax`. Wraz ze wzrostem precyzji arytmetyki rośnie również wyliczona wartość MAX.

1.3.5 Wnioski

Wnioskiem płynącym z analizy otrzymanych wyników jest fakt, że zastosowany algorytm działa poprawnie, ponieważ zwraca wartości MAX zgodne ze stanem faktycznym. Wzrost wartości MAX wraz ze zwiększaniem się precyzji arytmetyki wynika z faktu, że im większa precyzja, tym więcej liczb możemy w danej arytmetyce zapisać. Wartości zawarte w pliku nagłówkowym `float.h` są takie same jak dla języka `Julia`.

2 Zadanie II

2.1 Metoda Kahana

Kahan zaproponował metodę polegającą na obliczaniu wartości *macheps* za pomocą następującego wzoru: $3 \cdot (\frac{4}{3} - 1) - 1$.

2.2 Problem

Używając języka `Julia` należało porównać wartości zwracane przez wzór podany przez Kahana z wartościami *macheps* dla zadanych arytmetyk (`Float16`, `Float32`, `Float64`).

2.3 Algorytm

```
1: function KAHAN-MACHEPS-METHOD
2:     return 3.0 · (4.0/3.0 − 1.0) − 1.0
3: end function
```

Zakładamy, że wszelkie obliczenia zawarte w powyższym algorytmie są wykonywane w zadanej arytmetyce.

2.4 Wyniki

Program wykorzystujący podany wyżej algorytm dał następujące wyniki:

Arytmetyka `Float16`
Wartość uzyskana metodą Kahana: -0.000977
Wartość domyślna dla arytmetyki `Float16`: 0.000977

Arytmetyka `Float32`
Wartość uzyskana metodą Kahana: $1.1920929 \cdot 10^{-7}$
Wartość domyślna dla arytmetyki `Float32`: $1.1920929 \cdot 10^{-7}$

Arytmetyka `Float64`
Wartość uzyskana metodą Kahana: $-2.220446049250313 \cdot 10^{-16}$
Wartość domyślna dla arytmetyki `Float64`: $2.220446049250313 \cdot 10^{-16}$

2.5 Obserwacje

Wartości zwrócone przez algorytm są takie same jak faktyczne wartości *macheps* z dokładnością do znaku, tzn. otrzymane wartości *macheps* dla arytmetyk `Float16` i `Float64` mają przeciwny znak (wartość ujemną) niż te zwracane przez funkcję `eps` języka `Julia`.

2.6 Wnioski

Na podstawie uzyskanych wyników można wysnuć wniosek, że rozumowanie przeprowadzone przez Kahana było w gruncie rzeczy poprawne, ale zastosowana przez niego metoda będzie działać w pełni poprawnie dla wszystkich typów arytmetyki, jeżeli na zaproponowany wzór nałożymy jeszcze wartość bezwzględną: $\mid 3 \cdot (\frac{4}{3} - 1) - 1 \mid$.

3 Zadanie III

3.1 Obliczanie δ

Zgodnie z założeniami arytmetyki IEEE754, wszystkie reprezentowane liczby rozmieszczone są w pewnych odległościach δ określonych dla poszczególnych przedziałów.

3.2 Problem

Używając języka `Julia` i arytmetyki `Float64` należało sprawdzić, że liczby w przedziale $[1; 2]$ są równomiernie rozmieszczone z krokiem $\delta = 2^{-52}$, czyli każdą liczbę z zadanego przedziału $[1; 2]$ można zapisać jako $x = 1 + kx$, gdzie $k \in \{1, 2, ..., 2^{-52} - 1\}$ oraz $\delta = 2^{-52}$. W następnej kolejności należało sprawdzić rozmieszczenie liczb w przedziałach $[1; \frac{1}{2}]$ oraz $[2; 4]$.

3.3 Algorytm

```
1: function NUM-SHIFT(down, delta, steps)
2:     for k ← 1 to steps do
3:         x ← down + k · delta
4:         print BITS(x)
5:     end for
6: end function
```

Opis parametrów:
down - dolny zakres przedziału
delta - odstęp pomiędzy liczbami

steps - liczba iteracji

Zakłada się, że wszystkie obliczenia są wykonywane w zadanej arytmetyce.

3.4 Wyniki

Przedział: $[1; 2]$
Wartość δ: 2^{-52}
Liczba iteracji: 8

0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000000111
0011111111110000000000000000000000000000000000000000000000001000

Przedział: $[\frac{1}{2}; 1]$
Wartość δ: 2^{-53}
Liczba iteracji: 8

0011111111100000000000000000000000000000000000000000000000000001
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000011
0011111111100000000000000000000000000000000000000000000000000100
0011111111100000000000000000000000000000000000000000000000000101
0011111111100000000000000000000000000000000000000000000000000110
0011111111100000000000000000000000000000000000000000000000000111
0011111111100000000000000000000000000000000000000000000000001000

Przedział: $[2; 4]$
Wartość δ: 2^{-51}
Liczba iteracji: 8

0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
0100000000000000000000000000000000000000000000000000000000000101
0100000000000000000000000000000000000000000000000000000000000110
0100000000000000000000000000000000000000000000000000000000000111
0100000000000000000000000000000000000000000000000000000000001000

(Przedstawiono 8 pierwszych wyników dla każdego przedziału)

3.5 Obserwacje

W wyniku działania algorytmu otrzymano kolejne możliwe do reprezentacji liczby zmiennopozycyjne dla zadanych przedziałów. Zaobserwowano również następującą wartość δ dla poszczególnych przedziałów:

- $[1; 2]$: $\delta = 2^{-52}$,
- $[\frac{1}{2}; 1]$: $\delta = 2^{-53}$,
- $[2; 4]$: $\delta = 2^{-51}$.

3.6 Wnioski

Na podstawie uzyskanych wyników należy stwierdzić, że wzór $x = 1 + k\delta$ oraz wartość $\delta = 2^{-52}$ rzeczywiście opisują właściwe rozmieszczenie liczb w przedziale $[1; 2]$. Ponadto, wyznaczone wartości δ dla pozostałych dwóch przedziałów pozwalają wnioskować, iż wraz z oddalaniem się od wartości 0.0 odstępy pomiędzy liczbami są większe. Jednocześnie, w każdym z badanych przedziałów liczb jest dokładnie tyle samo, ponieważ zauważamy, że cecha dla wszystkich liczb z danego przedziału jest taka sama, a zmianie ulega jedynie mantysa.

4 Zadanie IV

4.1 Problem nieodwracalności dzielenia

W zadaniu należało zbadać problem nieodwracalności dzielenia w arytmetyce `Float64` dla języka `Julia`.

4.2 Problem

Zgodnie z poleceniem zadnia, należało eksperymentalnie, stosując arytmetykę `Float64`, 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.

4.3 Algorytm

```
1: function FIND-SMALLEST-NUM
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
```

Zakłada się, że wszystkie obliczenia są wykonywane w zadanej arytmetyce. Powyższa procedura zwraca najmniejszą liczbę spełniającą zadaną zależność.

4.4 Wyniki

Otrzymana wartość: 1.000000057228997.

4.5 Obserwacje

W wynik działa programu otrzymano najmniejszą możliwość wartość spełniającą zadaną zależność $fl(x \cdot fl(\frac{1}{x})) \neq 1$.

4.6 Wnioski

Fakt, iż dla arytmetyki `Float64` nie zachodzi równość $x \cdot \frac{1}{x} = 1$ wynika z błędów zaokrągleń, które powstają podczas wykonywania działań wykonywanych przy użyciu tej arytmetyki.

5 Zadanie V

5.1 Obliczanie iloczynu skalarnego

W zadaniu 5. należało przeprowadzić eksperyment polegający na obliczeniu 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 Problem

Używając języka `Julia` należało przeanalizować działanie oraz wyniki zwracane przez cztery różne algorytmy obliczające iloczyn skalarny wektorów X i Y.

5.3 Algorytm

(a) "w przód" - stosując algorytm $\sum_{i=1}^{n} x_i y_i$,

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

(b) "w tył" - stosując algorytm $\sum_{i=n}^{1} + x_i y_i$

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

(c) od największego do najmniejszego (dodając liczby dodatnie w porządku malejącym, dodając liczby ujemne w porządku rosnącym i na koniec dodając do siebie obliczone sumy częściowe

(d) odwrotnie niż w punkcie **(c)**.

5.4 Wyniki

W wyniku działania powyższych programów uzyskano następujące wyniki dla poszczególnych rozwiązań:

(a) Arytmetyka `Float32`: -0.4999443
Arytmetyka `Float64`: $1.0251881368296672 \cdot 10^{-10}$

(b) Arytmetyka `Float32`: -0.4543457
Arytmetyka `Float64`: $-1.5643308870494366 \cdot 10^{-10}$

(c) Arytmetyka `Float32`: -0.5
Arytmetyka `Float64`: 0.0

(d) Arytmetyka `Float32`: -0.5
Arytmetyka `Float64`: 0.0

Prawidłowy wynik iloczynu skalarnego dla wektorów X i Y to $-1.00657107000000 \cdot 10^{-11}$ (wynik z dokładnością do 15 cyfr znaczących).

5.5 Obserwacje

Wyniki dla poszczególnych arytmetyk oraz sposób rozwiązania zadania są różne. Żaden z uzyskanych wyników nie jest zgodny z rzeczywistym wynikiem mnożenia skalarnego wektorów X oraz Y.

5.6 Wnioski

Wnioskiem z uzyskanych wyników jest fakt, że kolejność wykonywania działań na liczbach zmiennopozycyjnych ma duże znaczenie dla uzyskanego wyniku. Wraz ze wzrostem precyzji arytmetyki rośnie również precyzja uzyskanego wyniku, aczkolwiek nawet dla obliczeń w arytmetyce `Float64` nie udało się uzyskać dokładnej wartości. Można zauważyć również, że na wielkość generowanego błędu wpływa rząd wielkości dodawanych do siebie liczb, np. jeżeli do liczby a dodamy liczbę b mniejszą od a o kilkanaście rzędów wielości, to wtedy wygenerujemy względnie niewielki błąd.

6 Zadanie VI

6.1 Obliczanie wartości funkcji

W zadaniu 6. należało przy użyciu języka `Julia` policzyć w arytmetyce `Float64` wartości dwóch zadanych funkcji rzeczywistych.

6.2 Problem

Zgodnie z poleceniem zadania, używając arytmetyki `Float64`, należało policzyć wartości dla funkcji $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla wartości $x \in \{8^{-1}, 8^{-2}, 8^{-3}, ...\}$.

6.3 Algorytm

Zostały obliczone wartości dla funkcji $f(x)$ i $g(x)$ za pomocą programów w języku `Julia`.

6.4 Wyniki

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

Podczas działania programu obliczono wartości funkcji f i g dla 200 pierwszych argumentów.

6.5 Obserwacje

Funkcje f i g są sobie równe. Dla kilku pierwszych wartości obliczone przez program wartości rzeczywiście są zbliżone, jednak wraz ze zmniejszaniem się wartości argumentu x widzimy co raz większe rozbieżności w wynikach. Funkcja f już dla argumentu $x = 8^{-9}$ osiągnęła wartość 0.0. Funkcja g osiągnęła wartość 0.0 dopiero dla argumentu $x = 8^{-179}$.

6.6 Wnioski

Można wykazać, że $\lim_{x\to 0} f(x) = \lim_{x\to 0} g(x) = 0$. Oznacza to, że funkcja f i g "zbliżają się" do wartości 0 dla kolejnych argumentów, ale nigdy tej wartości nie osiągną. Otrzymane wyniki stoją w sprzeczności z tym faktem, jednak nie są one sprzeczne z zasadą działania arytmetyki `Float64`. Zauważmy, że w przypadku funkcji f wartość pierwiastka jest bardzo zbliżona do wartości 1.0, zatem odejmowane są od siebie bardzo bliskie liczby w wyniku czego, dochodzi do utraty cyfr

znaczących. W przypadku funkcji g nie jest wykonywane żadne odejmowanie przez co dokładność wyników jest większa a wszelkie uzyskane błędy wynikają z precyzji arytmetyki. Przekształcenie funkcji f do postaci g jest jednym ze sposób uniknięcia problemów wynikających z utraty cyfr znaczących jak miało to miejsce w przypadku wartości uzyskanych dla funkcji f.

7 Zadanie VII

7.1 Obliczanie pochodnej funkcji

W zadaniu 7. należało, używając języka `Julia`, policzyć pochodną funkcji w arytmetyce `Float64` za pomocą zadanego wzoru, a następnie obliczyć błąd względny uzyskanych przybliżeń.

7.2 Problem

Przybliżoną wartość pochodnej funkcji f można obliczyć za pomocą następującego wzoru:

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

Obliczenia należało wykonać w arytmetyce `Float64`. Dana była funkcja $f(x) = \sin(x) + \cos(3x)$ oraz punkt $x_0 = 1$, w którym liczona była pochodna funkcji f. Należało również obliczyć błąd zadany wzorem $\mid f^{'}(x) - \tilde{f}^{'}(x) \mid$ dla $h = 2^{-n}$, gdzie $n \in \{0, 1, 2, ..., 54\}$.

7.3 Algorytm

Używając języka `Julia` i arytmetyki `Float64` zostały obliczone wartości $\tilde{f}^{'}(x)$ za pomocą podanego wcześniej wzoru. Obliczone zostały również wartości $f^{'}(x)$, aby możliwym było obliczone wartości błędu dla poszczególnych przybliżeń wartości pochodnej funkcji f.

7.4 Wyniki

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

7.5 Obserwacje

Na podstawie analizy otrzymanych wyników zauważamy, że dla pierwszych 29 wartości h otrzymujemy dosyć dokładne przybliżenia dla pochodnej funkcji f. Najdokładniejsze przybliżenie otrzymano dla $h = 2^{-28}$. Błąd wyniósł wtedy $4.802855890773117 \cdot 10^{-9}$. Jest to najmniejsza uzyskana wartość błędu. Wraz z dalszym zmniejszaniem się wartości h zauważamy, że wartości $1+h$ są coraz mniej dokładne, a dla najmniejszych h są równe 1.0.

7.6 Wnioski

Na podstawie poczynionych obserwacji zauważamy, że od pewnej wartości n ($h = 2^{-n}$), wartość h jest już na tyle mała, że dochodzi do znacznej utraty dokładności obliczeń. Dla $n < -28$ wyrażenie $1 + h$ dąży swoimi wartościami do 1.0, a liczony błąd zaczyna rosnąć. Wnioskiem płynącym z obserwacji jest fakt, iż należy unikać odejmowania od siebie wartości znacząco różniących się swoim wykładnikiem, ponieważ wpływa to na dokładność otrzymanych wyników. Na zaburzenie dokładności wyniku (utratę cyfr znaczących) wpływa również odejmowanie od siebie bardzo bliskich sobie wartości (dla bardzo małych h).