Sprawozdanie z listy nr 1 - Obliczenia Naukowe

Zadanie 1.

Opis Problemu

Pierwsza część zadania nr 1 polegała na iteracyjnym wyznaczeniu epsilonów maszynowych - ε_m - dla trzech typów zmiennopozycyjnych: Float16, Float32, Float64. Sprowadzała się ona do znalezienia maksymalnej liczby dla której wartość $fl(1.0 + macheps)$ wynosiła 1.0. Znając tę liczbę wystarczyło pomnożyć ją razy dwa, aby otrzymać najmniejsza liczbę spełniającą warunek $fl(1.0 + macheps) > 1.0$.

Rozwiązanie

W rozwiązaniu przyjąłem początkowy epsilon jako T(1.0). Następnie jest on dzielony przez 2 - co pozwalało sprawdzić czy epsilon mieści się w dostępnej liczbie mantysy przy wykładniku 2^0. Dokładna implementacja znajduję się w pliku [ex1.jl] w funkcji machine_epsilon_finder.

Wspomniana metoda empiryczna zwróciła następujące wyniki:

```
Type: Float16 Computed macheps: 0.000977 Built-in macheps: 0.000977
Are they equal? : true
Type: Float32 Computed macheps: 1.1920929e-7 Built-in macheps: 1.1920929e-7
Are they equal? : true
Type: Float64 Computed macheps: 2.220446049250313e-16 Built-in macheps: 2.220446049250313e-16
Are they equal? : true
```

Obserwacje i wnioski

Tak jak widać, dla wszystkich trzech typów znaleziony epsilon maszynowy równa się temu zwróconemu przez Julię (za pomocą komendy eps(T)). Potwierdza to poprawność zaimplementowanego algorytmu.

Eksperyment wyraźnie pokazuje, znaczące różnice precyzji - macheps jest bezpośrednio powiązany z precyzją ϵ równaniem.

$$|\delta| = \frac{|rd(x)-x|}{|x|} \leq \frac{1}{2}\frac{macheps}{|x|} \leq \frac{1}{2}\beta^{1-t} = \epsilon$$

Jest on wprost proporcjonalny, więc dla Float64 precyzja jest 10^9 większa w stosunku do Float32 i 10^{13} w stosunku do Float16.

Opis Problemu

Podobna logika zastosowana została w stosunku do drugiego podpunktu tego zadania - tym razem problemem było wyznaczenie liczby maszynowej η, czyli najmniejszej dodatniej liczby znormalizowanej - dla wspomnianych trzech typów zmiennoprzecinkowych.

Formalna definicja eta - jest to x takie, że

$$fl(x) > 0.0$$

Rozwiązanie

Raz jeszcze użyta została metoda iteracyjna - zmienna eta została zainicjalizowana jako T(1.0), a następnie była dzielona przez 2 (symulowanie mantysy) tak długo, aż jej wartość w danym typie zmiennoprzecinkowym nie wyniosła 0.0. Gdy to nastąpiło program zwracał wartość 2*eta - plik ex1.jl funkcja eta_finder.

Rezultat:

```
Type: Float16 Computed eta: 6.0e-8 Built-in eta: 6.0e-8
Are they equal? : true
Type: Float32 Computed eta: 1.0e-45 Built-in eta: 1.0e-45
Are they equal? : true
Type: Float64 Computed eta: 5.0e-324 Built-in eta: 5.0e-324
Are they equal? : true
```

Obserwacje

Wyniki są zgodne z teoretycznymi wartościami najmniejszych liczb denormalizowanych w standardzie IEE 754. Różnice między poszczególnymi wartościami są ogromne. Wynikają one bezpośrednio z liczby bitów przeznaczonych na wykładnik w każdym z typów: Float16 - 5, Float32 - 8, Float64 - 11. O to wartości, które każdy z nich może więc przechowywać: $2^{\pm 14}$, $2^{\pm 126}$, $2^{\pm 1023}$.

Wnioski

Liczba eta, która została wyliczona w tym zadaniu odpowiada najmniejszej liczbie denormalizowanej. Jest ona tożsama z przedstawioną na wykładzie liczbą MIN_{sub}.

Bezwątpliwie więc w kontekście obliczeń naukowych standard, który młody programista powinien wybrać to Float64. Dla pozostałych dwóch, overflow zaokrągla do zera zbyt duże liczby, co może mieć wpływ na wyniki. Tylko Float64 posiada na tyle szeroką mantysę by zapewnić wystarczającą dokładność liczb denormalizowanych.

Opis Problemu

W ostatniej już części zadania pierwszego należało najpierw odpowiedzieć na pytanie:

Co zwracają funkcje floatmin(Float32) i floatmin(Float64) i jaki jest związek zwracanych wartości z liczbą MIN_{nor}*?*

Funkcja floatmin zwracaja najmniejszą istniejącą liczbę znormalizowaną danego typu zmiennoprzecinkowego. Jest ona wprost tożsama z wprowadzoną na wykładzie liczbą M_{nor}.

```
Funkcja floatmin dla Float32: 1.1754944e-38
Funkcja floatmin dla Float64: 2.2250738585072014e-308
```

Kolejny już raz różnica jest znacząca. Wpływ na to ma dwukrotnie większa liczba przechowywanych bitów. Kiedy użytkownikowi zależy na dokładności, niewątpliwie powinien unikać korzystania z typów Float32 i Float16.

Rozwiązanie

Następnie należało napisać iteracyjny kod wyznaczający liczbę MAX dla standardowych trzech typów. Liczba MAX to po prostu maksymalna liczba skończona dla danego typu T.

Tym razem natomiast początkowa wartość nie jest dzielona tylko mnożona razy dwa tak długo aż funkcja isFinite(2*x) zwraca true. W przeciwnym wypadku zwracamy x (funkcja max_finder w ex1.jl)

Wyniki:

```
Type: Float16 Computed max: 3.277e4 Built-in max: 6.55e4
Are they equal? : false
Type: Float32 Computed max: 1.7014118e38 Built-in max: 3.4028235e38
Are they equal? : false
Type: Float64 Computed max: 8.98846567431158e307 Built-in max: 1.7976931348623157e308
Are they equal? : false
```

Obserwacje

Skąd pojawiają się różnice w wartościach?

Wartość początkowa to T(1.0) - odpowiada ona w standardzie wartości $1.0 * 2^{0}$.

Mnożąc razy dwa otrzymamy ostatecznie $1.0 * 2^{c_{max}}$. Różnica wynika więc z możliwości wstawienia .111111… po przecinku.

Wartość MAX jest więc niemal dwukrotnie większa niż ta którą obliczyliśmy.

Wnioski

Funkcja max_finder zwraca więc największą potęgę dwójki, która nie powoduje nadmiaru, a nie rzeczywistą wartość floatmax. Eksperyment potwierdził, też kluczowość wykładnika dla MAX różnych typów.

Podsumowując zadanie nr 1. Bezsprzecznie typ Float64 jest najlepszym z testowanych tutaj typów. Oferuję najlepszą precyzję, a przede wszystkim szeroki zakres - od MIN_{sub} po MIN_{nor}, aż po MAX.

Zadanie 2.

Opis Problemu

Celem zadania było potwierdzenie koncepcji Johna Kahana:

$$\epsilon_{m} = 3 * (\frac{4}{3} - 1) - 1$$

dla trzech typów zmiennoprzecinkowych.

Rozwiązanie

Implementacja jest prosta - jedyną trudnością tu występującą było pamiętanie o użyciu T(x) przed każdą z liczb.

Wyniki:

```
For type: Float16
Calculated hypothesis equals: -0.000977
Is it equal to actual?: false
For type: Float32
Calculated hypothesis equals: 1.1920929e-7
Is it equal to actual?: true
For type: Float64
Calculated hypothesis equals: -2.220446049250313e-16
Is it equal to actual?: false
```

Obserwacje

Obserwacje były początkowo zaskakujące - tylko dla Float32 rezultat eksperymentu zgadzał się z oczekiwaną wartością. Dla Float16 i Float64 wartość absolutna była prawidłowa, natomiast znak przed liczbą się nie zgadzał.
Odpowiedzią na pytanie dlaczego tak się stało jest sposób zaokrąglania w poszczególnych typach. W przypadku Float32 wartość $\frac{4}{3}$ została zaokrąglona w górę - przez co zwrócony wynik był poprawny.
Natomiast dla Float16 i Float64 w tym przypadku nastąpiło zaokrąglenie w dół. przez co $rd(\frac{4}{3} - 1) < \frac{1}{3}$.

Wnioski

Eksperyment dowiódł dwóch rzeczy:

- długość mantysy może spowodować różne zaokrąglenia, a co za tym idzie wpływa ona znacząco na zwracany wynik
- w sensie wartości absolutnych hipoteza Johna Kahana została udowodniona - $3 * (\frac{4}{3} - 1) - 1$ faktycznie opisuje epsilon maszynowy.

Zadanie 3.

Opis Problemu

Celem zadania było eksperymentalne sprawdzenie, jak liczby zmiennoprzecinkowe są rozmieszczone w różnych przedziałach.

Rozwiązanie

Pierwsza część polegała na iteracyjnej weryfikacji czy liczby w przedziale [1,2] są rozmieszczone równomiernie z krokiem $\delta = 2^{-52}$. Została do tego użyta funkcja analyse_gap w pliku ex3.jl. Jej implementacja jest prosta - do początkowej wartości 1.0 dodawany jest za każdym przejściem pętli δ, tak długo aż wartość nie osiągnie 2.0.

Obserwacje

Wyniki potwierdziły teoretyczną hipotezę przedstawioną w poleceniu zadania - faktycznie w całym przedziale $[1, 2)$ dodanie δ daje wartość równą nextfloat(x).

Dzieję się tak ponieważ, w tym przedziale wykładnik jest stały i wynosi 2^0, a zmiana liczby polega jedynie na dodawaniu 2^{-52} czyli jedynki do najmniej znaczącego bitu. Każda liczba maszynowa w tym przedziale ma więc postać:

$$x = 1 + k\delta \; dla \; k = 1, 2, \ldots, 2^{51}$$

Dla pozostałych przedziałów również przeprowadziłem analizę kroków. Tym razem po za funkcją nextfloat skorzystałem również z bitstring.

Wyniki prezentowały się w następujący sposób:
For interval: [0.5, 1.0]

```
The difference between 0.5 and 0.5000000000000001 equals: 1.1102230246251565e-16

Bit version of 0.5

0011111111100000000000000000000000000000000000000000000000000000

Bit version of nextfloat0.5000000000000001

0011111111100000000000000000000000000000000000000000000000000001

The difference between 1.0 and 1.0000000000000002 equals: 2.220446049250313e-16

Bit version of 1.0

0011111111110000000000000000000000000000000000000000000000000000

Bit version of nextfloat1.0000000000000002

0011111111110000000000000000000000000000000000000000000000000001

The difference between 0.5 and 0.5000000000000001 equals: 1.1102230246251565e-16

Bit version of 0.5

0011111111100000000000000000000000000000000000000000000000000000

Bit version of nextfloat0.5000000000000001

0011111111100000000000000000000000000000000000000000000000000001

For interval: [2.0, 4.0]
```

The difference between 2.0 and 2.0000000000000004 equals: 4.440892098500626e-16

Bit version of 2.0

0100000000000000000000000000000000000000000000000000000000000000

Bit version of nextfloat2.0000000000000004

0100000000000000000000000000000000000000000000000000000000000001

The difference between 4.0 and 4.000000000000001 equals: 8.881784197001252e-16

Bit version of 4.0

0100000000010000000000000000000000000000000000000000000000000000

Bit version of nextfloat4.000000000000001

0100000000010000000000000000000000000000000000000000000000000001

The difference between 2.0 and 2.0000000000000004 equals: 4.440892098500626e-16

Bit version of 2.0

0100000000000000000000000000000000000000000000000000000000000000

Bit version of nextfloat2.0000000000000004

0100000000000000000000000000000000000000000000000000000000000001

Widać, że w każdym przypadku dodanie epsilonu jest równoważne z dodaniem jedynki na najmniej znaczącym bicie. Natomiast występuje znacząca różnica wartości. Z czego to wynika?
Weźmy dwie liczby - jedną z wykładnikiem 2 a drugą z 3. Chociaż operacja dodania jedynki na końcu mantysy jest ta sama to wartość tej jedynki jest inna:

$$0.0000...001 * 2^2 \neq 0.0000...001 * 2^3$$

Wnioski:

- eksperyment dowodzi, że liczby zmiennoprzecinkowe są rozmieszczone nieliniowo na osi x. Wielkość kroku jest proporcjonalna do rzędu wielkości liczby x. Odstępy są stałe, gdy nie zmienia się wartość wykładnika liczby czyli w przedziale $[2^E, 2^{E+1})$. Wartość kroku opisywana jest przez $2^{-52} * 2^E$.

- w ogólności w przedziale $[2^E, 2^{E+1}]$ liczba może być przedstawiona jako: $x = 2^E + k * \frac{2^E}{\delta}$

Zadanie 4.

Opis problemu

Celem zadania było eksperymentalne znalezienie w arytmetyce Float64 liczby x w przedziale 1 < x < 2, dla której naruszone jest prawo odwrotności mnożenia, czyli:

$$x * fl(\frac{1}{x}) \neq 1$$

Rozwiązanie

Program w języku Julia iteruje przez wszystkie reprezentowalne liczby zmiennoprzecinkowe w przedziale $[1 + \epsilon_m, 2)$ z krokiem δ i sprawdza warunek $x * fl(\frac{1}{x}) \neq 1$.

Wynik:

```
For type: Float64

The smalles number found: 1.000000057228997
```

Obserwacje

Dlaczego tak się dzieje?

W działaniu tym występują dwa zaokrąglenia. Mimo tego, że x jest reprezentowalny, gdyż jest wielokrotnością δ plus jeden, wartość $\frac{1}{x}$ nie jest reprezentowalna, więc musi wystąpić zaokrąglenie w górę, bądź w dół. Kolejne przybliżenia następują, gdy wykonujemy mnożenie otrzymanej wartości razy x. Dla otrzymanej wartości x otrzymany błąd jest na tyle duży, że powoduje on nieprawidłowy wynik.

Wnioski

Eksperyment ten udowadnia, że w arytmetyce maszynowej nie można zakładać matematycznej tożsamości $x * \frac{1}{x} = 1$. Zawsze należy brać pod uwagę błąd, który jest wynikiem wielokrotnego zaokrąglania. Jak widać dzieje się to szybko bo już dla wartości 1.000000057228997.

Zadanie 5.

Opis problemu

Problemem jest obliczenie iloczynu skalarnego dwóch wektorów x i y za pomocą różnych algorytmów sumowania i przy użyciu dwóch precyzji: Float32 i Float64. Po wykonaniu odpowiednich algorytmów należało porównać otrzymane wyniki i wyciągnąć wnioski o arytmetyce w typach zmiennoprzecinkowych.

Rozwiązanie

Eksperyment przeprowadzony został za pomocą czterech implementacji w języku Julia. Każda z 4 funkcji wykonała się dwukrotnie - raz dla każdego typu. Poniżej znajduje się krótki opis każdej z funkcji. Dokładną implementacje można znaleźć w pliku ex5.jl.

Funkcje:

- first: proste sumowanie z wykorzystaniem pętli for
- second: podobnie jak dla first lecz z odwróconą kolejnością
- third: na wstępie wektory przechodzą proces konwersji na dany typ. Następnie tworzona jest tablica zawierająca termsy czyli: $x_i * y_i$. Kolejny krok to podzielenie tej tablicy na dwie - jedną zawierającą liczby dodatnie i drugą zawierającą liczby ujemne - a następnie sortowanie ich po wartości absolutnej. Na końcu obie są sumowane i dodawane do siebie.
- fourth: analogicznie do trzeciej funkcji lecz tu sortowanie jest na odwrót.

Wyniki:

Type: Float32

Result for function first is -0.4999443

```
Result for function second is -0.4543457

Sum of positive: 2.7595028e6 Sum of negative: -2.7595032e6

Result for function third is -0.5

Result for function fourth is -0.5

Type: Float64

Result for function first is 1.0251881368296672e-10

Result for function second is -1.5643308870494366e-10

Sum of positive: 2.7595029196180594e6 Sum of negative: -2.7595029196180594e6

Result for function third is 0.0

Result for function fourth is 0.0
```

Obserwacje

Dla typu Float32 wszystkie metody zawiodły całkowicie - wyniki mają rzędy wielkości o 10^8 większe od spodziewanego rezultatu. Jest to spowodowane niewystarczającą liczbą bitów przeznaczonych na mantysę. Wektory x i y zawierały w sobie liczby różniące się o kilka rzędów wielkości. W konsekwencji podczas dodawania wiele informacji było traconych podczas przesuwania mniejszej liczby "w prawo" na mantysie.

W przypadku typu Float64 niewątpliwie wyniki były bardziej dokładne. Rząd ich wielkości to 10^{-10} (dla pierwszych dwóch funkcji). Natomiast dla sumowania w przód błąd wynosił 200% - wynik pomimo podobnej wielkości był dodatni. Sugeruje to duże straty informacji podczas dodawania. Metoda "w tył" była dokładniejsza - błąd był na poziomie 55% - jednakże wciąż nie udało się zniwelować problemu katastrofalnego zaokrąglenia.

Takowy rezultat nie powinien być jednak zaskoczeniem - w przypadku obu metod dodawanie jest tak naprawdę losowe i zależy od ułożenia wektorów. Występuje więc duża szansa dodawania dwóch liczb o znacząco różnym rzędzie wielkości.

Natomiast w przypadku sposobu c i d program zwrócił 0. Jest to zachowanie niespodziewane - podczas wstępnej analizy zadania metody te były najbardziej obiecujące. Teoretycznie powinny one niwelować dodawanie liczb znacznie się od siebie różniących. Jednakże w tym przypadku nastąpiły dwa problemy:

- po pierwsze wektor produktu wektorów x i y składał się z liczb o ogromnych różnicach. Ich ustawienie choć pomocne nie było wstanie zniwelować powstałego błędu.
- Jak zostało pokazane w sekcji wyniki tablica liczb dodatnich i ujemnych zwróciła bardzo podobny wynik, lecz różniący się znakiem. Z tego powodu nastapiło odejmowanie liczb do siebie podobnych - zagrożenia z tym związane zostały pokazane w kolejnym zadaniu.

Wnioski

Eksperyment wykazał, że typ Float32 całkowicie nie nadaje się do precyzyjnych obliczeń. Dla każdej z zaimplementowanych metod zwrócił on wynik pozbawiony sensu. Co więcej dodawanie liczb w sposób losowy - w przód lub w tył - niesie ze sobą zagrożenie utraty informacji, związanej z dodaniem dwóch liczb o znacząco różniącym się rzędzie wielkości.

W przypadku iloczynów skalarnych naturalnie obarczonych utratą precyzji należy korzystać z bardziej zaawansowanych algorytmów. Nawet te teoretycznie lepsze - c i d nie są wystarczające dla tego przypadku.

Zadanie 6.

Opis problemu

Celem tego eksperymentu była analiza stabilności numerycznej dwóch matematycznie tożsamych funkcji w arytmetyce Float64 dla argumentów $x -> 0$.

Otrzymane wyniki:

```
julia> include("./ex6.jl")
| 1 | 0.125              | 0.0077822185373186414 | 0.0077822185373187065 | 8.359072677949159e-15 |
| 2 | 0.015625           | 0.00012206286282867573 | 0.00012206286282875901 | 6.822736862306436e-13 |
| 3 | 0.001953125        | 1.9073468138230965e-6 | 1.907346813826566e-6 | 1.818991138267678e-12 |
| 4 | 0.000244140625     | 2.9802321943606103e-8 | 2.9802321943606116e-8 | 4.440892164675074e-16 |
| 5 | 3.0517578125e-5    | 4.656612873077393e-10 | 4.6566128719931904e-10 | 2.3283064370807974e-10 |
| 6 | 3.814697265625e-6 | 7.275957614183426e-12 | 7.275957614156956e-12 | 3.637978807104948e-12 |
| 7 | 4.76837158203125e-7 | 1.1368683772161603e-13 | 1.1368683772160957e-13 | 5.684341886081125e-14 |
| 8 | 5.960464477539063e-8 | 1.7763568394002505e-15 | 1.7763568394002489e-15 | 8.881784197001026e-16 |
| 9 | 7.450580596923828e-9 | 0.0                    | 2.7755575615628914e-17 | 1.0                |
```

Obserwacje

Funkcja f zawodzi dla i=9 z powodu odejmowania. Dla:

$$x -> 0 \sqrt{x + 1} = 0$$

Wtedy odejmujemy ze sobą dwie liczby podobne więc tracimy dużo “informacji”. Oznacza to, że o wartości wyniku zadecydują bity mniej znaczące, co może powodować nieprawidłowe zaokrąglenie. Spójrzmy na definicję z wykładu:

Proces numeryczny jest niestabilny, gdy niewielkie błędy, popełnione w początkowym stadium procesu kumulują się w kolejnych stadiach, powodując poważną utratę dokładności obliczeń.

W przypadku funkcji f definicja ta jest niewątpliwie spełniona. Dla 8^{-9} błąd wynosi 100%.

Tymczasem funkcja g całkowicie unika odejmowania bliskich sobie liczb. Jest ona funkcją stabilną.

Wnioski

Ten eksperyment dowodzi, że w obliczeniach numerycznych algebraiczna tożsamość nie gwarantuje numerycznej wierności. Musimy aktywnie przekształcać wzory, aby wyeliminować operacje odejmowania, które prowadzą do utraty precyzji, gdy argumenty są bliskie zeru. Czyli zawsze powinniśmy poszukiwać funkcji stabilnych.

Zadanie 7.

Opis problemu

Celem tego eksperymentu była analiza wpływu kroku h na dokładność przybliżenia funkcji $f(x) = sin(x) + cos(3x)$ w punkcie $x_0 = 1.0$, przy użyciu wzoru różnic skończonych:

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

Rozwiązanie

Eksperyment polegał na iteracyjnym zmniejszaniu kroku $h = 2^{-n}$ i obserwacji błędu bezwzględnego w stosunku do dokładnej wartości pochodnej która wynosi: 0.11694228168853815.

Wyniki:

```
Eksperyment: Przybliżenie pochodnej f(x) = sin(x) + cos(3x) w x0 = 1.0
Exact value of  f'(1.0): 0.11694228168853815
|  0 | 1.0000000000000000e+00 | 2.0179892252685967e+00 | 1.9010469435800585e+00 | 2.0000000000000000e+00 | 1.6256284007210247e+03 |
|  1 | 5.0000000000000000e-01 | 1.8704413979316472e+00 | 1.7534991162431091e+00 | 1.5000000000000000e+00 | 1.4994569038026341e+03 |
|  2 | 2.5000000000000000e-01 | 1.1077870952342974e+00 | 9.9084481354575926e-01 | 1.2500000000000000e+00 | 8.4729389510695250e+02 |
|  3 | 1.2500000000000000e-01 | 6.2324127929758166e-01 | 5.0629899760904351e-01 | 1.1250000000000000e+00 | 4.3294776730756001e+02 |
|  4 | 6.2500000000000000e-02 | 3.7040006620351917e-01 | 2.5345778451498102e-01 | 1.0625000000000000e+00 | 2.1673750576377125e+02 |
|  5 | 3.1250000000000000e-02 | 2.4344307439754687e-01 | 1.2650079270900871e-01 | 1.0312500000000000e+00 | 1.0817369977945917e+02 |
|  6 | 1.5625000000000000e-02 | 1.8009756330732785e-01 | 6.3155281618789694e-02 | 1.0156250000000000e+00 | 5.4005515119840283e+01 |
|  7 | 7.8125000000000000e-03 | 1.4849139537109579e-01 | 3.1549113682557639e-02 | 1.0078125000000000e+00 | 2.6978363366113335e+01 |
|  8 | 3.9062500000000000e-03 | 1.3270911428051591e-01 | 1.5766832591977753e-02 | 1.0039062500000000e+00 | 1.3482576502116519e+01 |
|  9 | 1.9531250000000000e-03 | 1.2482369294070850e-01 | 7.8814112521703450e-03 | 1.0019531250000000e+00 | 6.7395736925687375e+00 |
| 10 | 9.7656250000000000e-04 | 1.2088247681106168e-01 | 3.9401951225235265e-03 | 1.0009765625000000e+00 | 3.3693503030988974e+00 |
| 11 | 4.8828125000000000e-04 | 1.1891225046883847e-01 | 1.9699687803003130e-03 | 1.0004882812500000e+00 | 1.6845650280256121e+00 |
| 12 | 2.4414062500000000e-04 | 1.1792723373901026e-01 | 9.8495205047210987e-04 | 1.0002441406250000e+00 | 8.4225485961990421e-01 |
| 13 | 1.2207031250000000e-04 | 1.1743474961076572e-01 | 4.9246792222756852e-04 | 1.0001220703125000e+00 | 4.2112050074343360e-01 |
| 14 | 6.1035156250000000e-05 | 1.1718851362093119e-01 | 2.4623193239303731e-04 | 1.0000610351562500e+00 | 2.1055851556654823e-01 |
| 15 | 3.0517578125000000e-05 | 1.1706539714577957e-01 | 1.2311545724141837e-04 | 1.0000305175781250e+00 | 1.0527882256421311e-01 |
| 16 | 1.5258789062500000e-05 | 1.1700383928837255e-01 | 6.1557599834394239e-05 | 1.0000152587890625e+00 | 5.2639301154004822e-02 |
| 17 | 7.6293945312500000e-06 | 1.1697306045971345e-01 | 3.0778771175299369e-05 | 1.0000076293945312e+00 | 2.6319625999153124e-02 |
| 18 | 3.8146972656250000e-06 | 1.1695767106721178e-01 | 1.5389378673624776e-05 | 1.0000038146972656e+00 | 1.3159807087236896e-02 |
| 19 | 1.9073486328125000e-06 | 1.1694997636368498e-01 | 7.6946751468298658e-06 | 1.0000019073486328e+00 | 6.5798914094422380e-03 |
| 20 | 9.5367431640625000e-07 | 1.1694612901192158e-01 | 3.8473233834324105e-06 | 1.0000009536743164e+00 | 3.2899335705449109e-03 |
| 21 | 4.7683715820312500e-07 | 1.1694420524872839e-01 | 1.9235601902423127e-06 | 1.0000004768371582e+00 | 1.6448799890577527e-03 |
| 22 | 2.3841857910156250e-07 | 1.1694324295967817e-01 | 9.6127114002086955e-07 | 1.0000002384185791e+00 | 8.2200477546786775e-04 |
| 23 | 1.1920928955078125e-07 | 1.1694276239722967e-01 | 4.8070869151928264e-07 | 1.0000001192092896e+00 | 4.1106491559621955e-04 |
| 24 | 5.9604644775390625e-08 | 1.1694252118468285e-01 | 2.3949614469387370e-07 | 1.0000000596046448e+00 | 2.0479859058312474e-04 |
| 25 | 2.9802322387695312e-08 | 1.1694239825010300e-01 | 1.1656156484463054e-07 | 1.0000000298023224e+00 | 9.9674440383400761e-05 |
| 26 | 1.4901161193847656e-08 | 1.1694233864545822e-01 | 5.6956920069239914e-08 | 1.0000000149011612e+00 | 4.8705155438080039e-05 |
| 27 | 7.4505805969238281e-09 | 1.1694231629371643e-01 | 3.4605178278468429e-08 | 1.0000000074505806e+00 | 2.9591673583584765e-05 |
| 28 | 3.7252902984619141e-09 | 1.1694228649139404e-01 | 4.8028558907731167e-09 | 1.0000000037252903e+00 | 4.1070311109244062e-06 |
| 29 | 1.8626451492309570e-09 | 1.1694222688674927e-01 | 5.4801788884617508e-08 | 1.0000000018626451e+00 | 4.6862253834396319e-05 |
| 30 | 9.3132257461547852e-10 | 1.1694216728210449e-01 | 1.1440643366000813e-07 | 1.0000000009313226e+00 | 9.7831538779717027e-05 |
| 31 | 4.6566128730773926e-10 | 1.1694216728210449e-01 | 1.1440643366000813e-07 | 1.0000000004656613e+00 | 9.7831538779717027e-05 |
| 32 | 2.3283064365386963e-10 | 1.1694192886352539e-01 | 3.5282501276157063e-07 | 1.0000000002328306e+00 | 3.0170867856099990e-04 |
| 33 | 1.1641532182693481e-10 | 1.1694145202636719e-01 | 8.2966217096469563e-07 | 1.0000000001164153e+00 | 7.0946295812356563e-04 |
| 34 | 5.8207660913467407e-11 | 1.1694145202636719e-01 | 8.2966217096469563e-07 | 1.0000000000582077e+00 | 7.0946295812356563e-04 |
| 35 | 2.9103830456733704e-11 | 1.1693954467773438e-01 | 2.7370108037771956e-06 | 1.0000000000291038e+00 | 2.3404800763738290e-03 |
| 36 | 1.4551915228366852e-11 | 1.1694335937500000e-01 | 1.0776864618478044e-06 | 1.0000000000145519e+00 | 9.2155416012669738e-04 |
| 37 | 7.2759576141834259e-12 | 1.1692810058593750e-01 | 1.4181102600652196e-05 | 1.0000000000072760e+00 | 1.2126582785875407e-02 |
| 38 | 3.6379788070917130e-12 | 1.1694335937500000e-01 | 1.0776864618478044e-06 | 1.0000000000036380e+00 | 9.2155416012669738e-04 |
| 39 | 1.8189894035458565e-12 | 1.1688232421875000e-01 | 5.9957469788152196e-05 | 1.0000000000018190e+00 | 5.1270993623881725e-02 |
| 40 | 9.0949470177292824e-13 | 1.1682128906250000e-01 | 1.2099262603815220e-04 | 1.0000000000009095e+00 | 1.0346354140789013e-01 |
| 41 | 4.5474735088646412e-13 | 1.1694335937500000e-01 | 1.0776864618478044e-06 | 1.0000000000004547e+00 | 9.2155416012669738e-04 |
| 42 | 2.2737367544323206e-13 | 1.1669921875000000e-01 | 2.4306293853815220e-04 | 1.0000000000002274e+00 | 2.0784863697590697e-01 |
| 43 | 1.1368683772161603e-13 | 1.1621093750000000e-01 | 7.3134418853815220e-04 | 1.0000000000001137e+00 | 6.2538901924797430e-01 |
| 44 | 5.6843418860808015e-14 | 1.1718750000000000e-01 | 2.4521831146184780e-04 | 1.0000000000000568e+00 | 2.0969174529616039e-01 |
| 45 | 2.8421709430404007e-14 | 1.1328125000000000e-01 | 3.6610316885381522e-03 | 1.0000000000000284e+00 | 3.1306313128803782e+00 |
| 46 | 1.4210854715202004e-14 | 1.0937500000000000e-01 | 7.5672816885381522e-03 | 1.0000000000000142e+00 | 6.4709543710569166e+00 |
| 47 | 7.1054273576010019e-15 | 1.0937500000000000e-01 | 7.5672816885381522e-03 | 1.0000000000000071e+00 | 6.4709543710569166e+00 |
| 48 | 3.5527136788005009e-15 | 9.3750000000000000e-02 | 2.3192281688538152e-02 | 1.0000000000000036e+00 | 1.9832246603763071e+01 |
| 49 | 1.7763568394002505e-15 | 1.2500000000000000e-01 | 8.0577183114618478e-03 | 1.0000000000000018e+00 | 6.8903378616492370e+00 |
| 50 | 8.8817841970012523e-16 | 0.0000000000000000e+00 | 1.1694228168853815e-01 | 1.0000000000000009e+00 | 1.0000000000000000e+02 |
| 51 | 4.4408920985006262e-16 | 0.0000000000000000e+00 | 1.1694228168853815e-01 | 1.0000000000000004e+00 | 1.0000000000000000e+02 |
| 52 | 2.2204460492503131e-16 | -5.0000000000000000e-01 | 6.1694228168853815e-01 | 1.0000000000000002e+00 | 5.2756135144659697e+02 |
| 53 | 1.1102230246251565e-16 | 0.0000000000000000e+00 | 1.1694228168853815e-01 | 1.0000000000000000e+00 | 1.0000000000000000e+02 |
| 54 | 5.5511151231257827e-17 | 0.0000000000000000e+00 | 1.1694228168853815e-01 | 1.0000000000000000e+00 | 1.0000000000000000e+02 |
```

Obserwacje

W eksperymencie widzimy, że początkowo błąd maleje, a potem gwałtownie rośnie. Dla n<28 błąd jest zdominowany przez błąd wynikający z samego wzoru przybliżającego pochodną. W miarę zmniejszenia h błąd obcięcia, który jest rzędu O(h). Im mniejsze h tym mniejszy błąd. Jednakże w pewnym momencie większe znaczenie odgrywa wspomniane w poprzednim zadaniu odejmowanie. Wartość h zbliża się do zera więc zaczynamy odejmować od siebie dwie podobne liczby - powoduje to destrukcje danych i sprawia, że mało znaczące bity decydują o wartości. Następnie dzielimy przez małe h, co również zwiększa błąd. Widać, że dla n=53 i n=54 błąd jest równy 100%.

Wnioski

Raz jeszcze eksperyment dowodzi, że w metodach numerycznych należy brać pod uwagę stabilność funkcji. Pomimo, iż teoretycznie dla mniejszych h wynik powinien być dokładniejszy (gdyż redukuje błąd wynikający z przybliżenia Sterlinga) to z powodu niestabilności jest on całkowicie niepoprawny.