Metody obliczeniowe w nauce i technice

Filip Twardy

8 March 2020

1 Wstep

Celem ćwiczenia było sprawdzenie jak liczby zmiennopozycyjne pojedynczej oraz podwójnej precyzji reprezentowane sa w architekturze komputera oraz na jakie błedy możemy spotkać nawet podczas prostej sumy liczb. Kod wyliczajacy wszystkie wartości nosi nazwe *lab.cpp*.

2 Wzory

Wzory użyte w sprawozdaniu:

Bład bezwzgledny $\Delta x = |x - x_0|$

Bład wzgledny $\delta = \frac{|x-x_0|}{x}$

Funkcja dzeta Riemanna $\zeta(s) = \sum_{k=1}^{n} \frac{1}{k^s}$

Funkcja eta Dirichleta $\eta(s) = \sum_{k=1}^{n} (-1)^{k-1} \frac{1}{k^s}$

W sprawozdaniu bede posługiwał sie wyłacznie nazwami badź odpowiednimi symbolami danego wzoru.

3 Sumowanie liczb pojedynczej precyzji

W tym zadaniu sprawdzaliśmy bład wzgledny oraz bezwzgledny prostego sumowania liczb zmiennopozycyjnych o pojedynczej precyzji.

Najpierw dokonałem sumowania 10^7 razy liczby 0.012. Jak wiemy wynik dla nas jest trywialny równy 120000. Wyniki doświadczenia:
$\Delta x = 0.122831$
$\delta = 14739.8$
Jak widzimy bład bezwzgledny jest ogromny spowodowany wielokrotnym powtarzaniem małego błedu.

Wykres pokazuje w jaki sposób rośnie bład wzgledny w trakcie sumowania (raportowane co 2500 kroków).

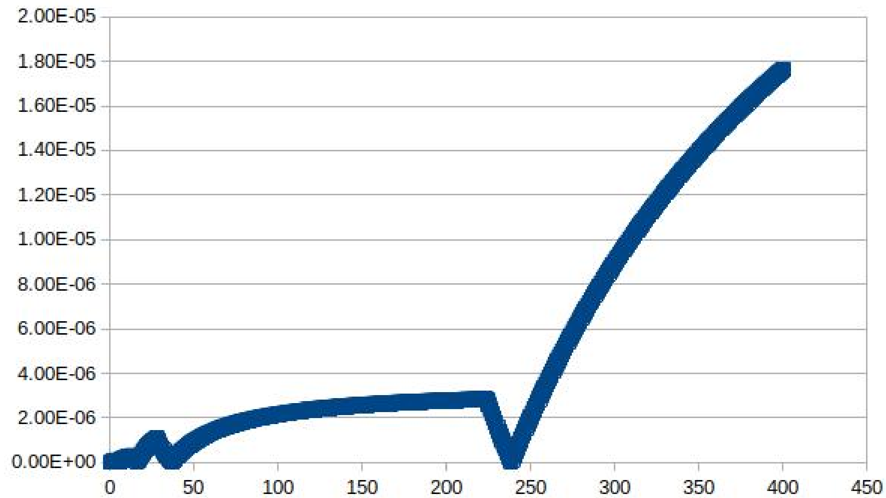

Figure 1: Wykres błedu wzglednego do ilości sum

Nastepnie zaimplementowałem rekurencyjny algorytm sumowania liczb, liczby sa tego samego rzedu co jak zakładam powinno zmniejszyć bład. Wyniki doświadczenia:
$\Delta x = 0$
$\delta = 0$
Jak widzimy bład dla tego przypadku zniknał. Bład zmalał ponieważ sumowanie liczb tego samego rzedu daje lepsze efekty niż sumowanie liczb znacznie różniacych sie przesunieciem dziesietnym.

Porównanie czasu działania obu algorytmów:
Pierwszy algorytm sumowania: $78528\mu s$
Drugi algorytm sumowania: $1051874\mu s$
Drugi algorytm jest wolniejszy ale dokładniejszy.

Algorytm rekurencyjny nie jest nieomylny dla danych wejściowych:
a = 12.3123123 n = 10^7
$\Delta x = 0.02$

4 Algorytm Kahana

W tym ćwiczeniu zaimplementowałem algorytm Kahana sumowania liczb, a nastepnie wyliczyłem błedy obliczeń. Algorym Kahana jest znacznie szybszy niż algorytm rekurencyjny oraz niewiele gorszy od klasycznego sumowania ponieważ wykonuje jedynie 3 dodatkowe operacje w pojedynczej iteracji. Bazuje na dodatkowym parametże **err** jest on wykożystywany do zmniejszenia błedu obliczeniowego tego algorytmu.

Wyniki doświadczenia:
$\Delta x = 0$
$\delta = 0$
czas działania = $118028 \mu s$
Algorytm Kahana jest 10 razy szybszy od algorytmu rekurencyjnego.

5 Sumy cześciowe

Ćwiczenie polegało na implementacji funkcji dzeta Riemanna $\zeta(s)$ oraz funkcji eta Dirichleta $\eta(s)$. Nastepnie porównania wyników dla pojedynczej oraz podówjnej precyzji. Pełne dane dostepne sa po właczeniu probramu *lab.cpp zad3()* sa one duże i nie umieszczam ich bezpośrednio w sprawozdaniu. Obliczenia dokonałem dla s = 2, 3.6667, 5, 7.2, 10 oraz n = 50, 100, 200, 500, 1000.

Analiza danych pokazuje, że bład wzgledny pomiedzy wynikami dla pojedynczej a podówjnej precyzji cześciej pojawia sie przy sumowaniu do przodu tzn. od k=1 do n.

6 Błedy zaokragleń i odwzorowanie logistyczne

Zadanie polegało na analizie odwzorowania logistycznego danego wzorem rekurencyjnym:
$x_{n+1} = rx_n(1 - x_n)$

Dokonałem wyboru $x_0 = 0.25, 0.5, 0.75$ oraz $2 \leq r \leq 4$
Diagramy bifurakcyjne otrzymanych wyników:

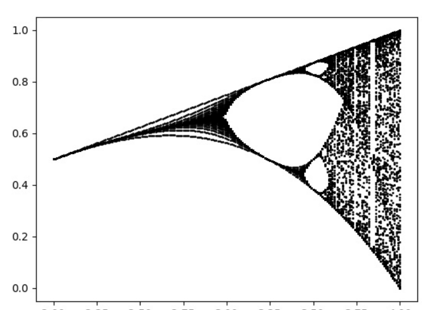

Figure 2: $x_0 = 0.25$

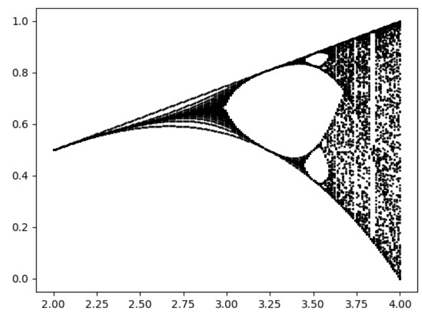

Figure 3: $x_0 = 0.5$

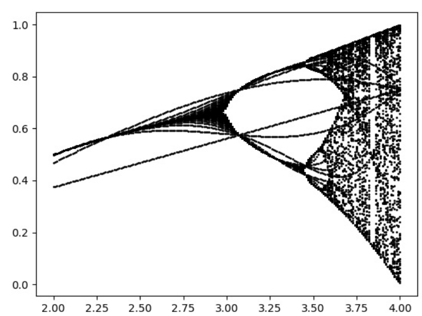

Figure 4: $x_0 = 0.75$

Analiza wyników pokazuje, że wraz ze wzrostem wartości parametru r, uzyskujemy wieksza różnice pomiedzy wartościami x.

Nastepnie przyjałem wartość $x_0 = 0.5$ oraz $3.75 \leq r \leq 3.8$ Wyników było dużo wiec nie umieszczam tabelki tutaj, natomiast sa one dostepne po odpowiednim uruchomieniu programy *lab.cpp* z odkomentowanymi liniami odpowiadajacymi wyliczaniu tych wartości.

Analiza pokazuje, że wraz ze wzrostem r, wartości x w kolejnych przejściach uzyskuja co raz bardziej rozbierzne wartości. Dla liczb zmiennopozycyjnych o podówjnej precyzji jest to jeszcze bardziej zauważalne.
Dla wartości $x_0 = 0.00005, 0.33, 0.5$ oraz r = 4 obliczyłem ile iteracji potrzeba aby $x = 0$. Wyniki:
$0.00005 \rightarrow 196$
$0.33 \rightarrow 1307$
$0.5 \rightarrow 2$

Wnioski sa nastepujace, ilość iteracji jest cieżko określić dla danej liczby, a same wartości ilości iteracji przyjmuja bardzo różne wartości, natomiast dla niektórym wartości przykładowo $x_0 = 0.4$ liczba iteracji jest tak duża, że po około 5 minutach pracy programu dalej nie została wyliczona.