AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody obliczeniowe w nauce i technice

Laboratorium 1

Jan Gołda, WIET 2018/2019

1. Wprowadzenie

Laboratorium miało na celu pokazanie problemów związanych z operacjami (w tym przypadku dodawania) na liczbach zmiennoprzecinkowych, oraz rozwiązań pozwalających na minimalizację lub całkowite pozbycie się tego problemu.

Według oryginalnego zadania mieliśmy tablicę którą następnie będziemy sumować, jednak zważywszy że każdy element tablicy jest taki sam, a ona sama nie wpływa na interesujące nas badanie pozwoliłem sobie na pominięcie tablicy, dzięki czemu kod jest znacznie krótszy.

2. Operacja sumowania liczb poprzez iterację

2.1. Kod wykorzystywany w testach

```
#define N …
#define a …

int main(){
    printf("Calculating %f * %d (iteration)\n", a, N);

    double sum = 0;
    for(int i = 0; i < N; i++)
        sum += a;

    printf("Calculated as sum:\t%f\n", sum);
    printf("Calculated as mul:\t%f\n", N*a);
    printf("Difference:\t\t%f\n", abs(sum - N*a));
    printf("Error:\t\t\t%.2f%%\n", abs(sum - N*a)/(N*a)*100);

    return 0;
}
```

2.2. Wyniki

N = 10000000 **a = 0.730270**	**Float**	Calculated as sum: 6386065.500000 Calculated as mul: 7302700.500000 Difference: **916635.000000** Error: **12.55%**
	Double	Calculated as sum: 7302700.281143 Calculated as mul: 7302700.500000 Difference: **0.218857** Error: **0.00%**
N = 10000000 **a = 0.492710**	**Float**	Calculated as sum: 4997944.000000 Calculated as mul: 4927100.000000 Difference: **70844.000000** Error: **1.44%**
	Double	Calculated as sum: 4927099.943161 Calculated as mul: 4927100.000000 Difference: **0.056839** Error: **0.00%**
N = 10000000 **a = 0.004356**	**Float**	Calculated as sum: 39962.527344 Calculated as mul: 43560.000000 Difference: **3597.472656** Error: **8.26%**
	Double	Calculated as sum: 43560.001068 Calculated as mul: 43560.000000 Difference: **0.001068** Error: **0.00%**

2.3. Wnioski

Pomiary pokazują że w przypadku liczb o pojedynczej precyzji mamy do czynienia ze znacznym błędem, nawet ponad 10%. Zastosowanie liczb o podwójnej precyzji co prawda znacznie redukuje błąd (o kilka rzędów wielkości), ale nie usuwa go całkowicie.

A zatem dodawanie liczb zmiennoprzecinkowych nie jest matematycznie poprawnym dodawaniem i nie powinno się go używać gdy dokładność jest istotna.

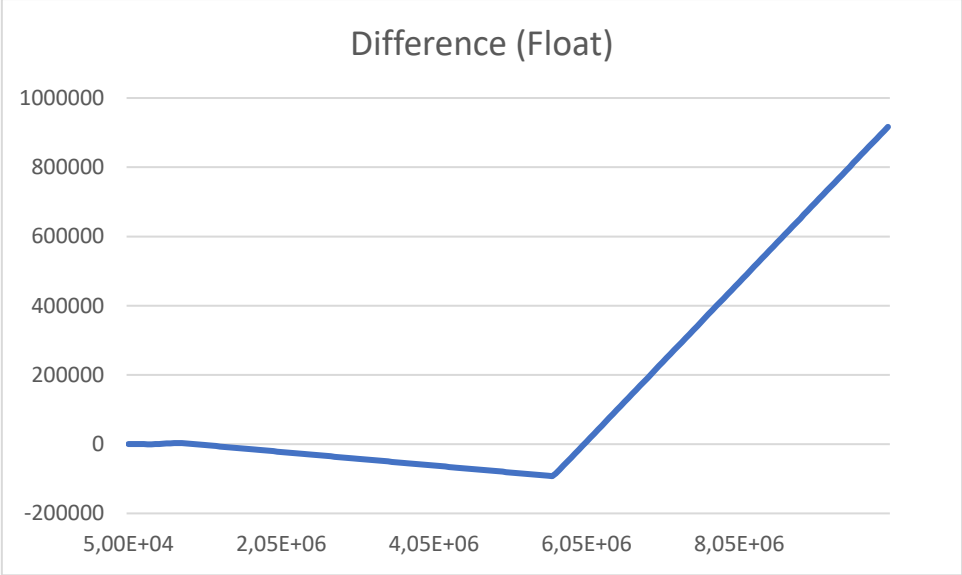
3. Analiza zmiany błędu na różnych etapach sumowania

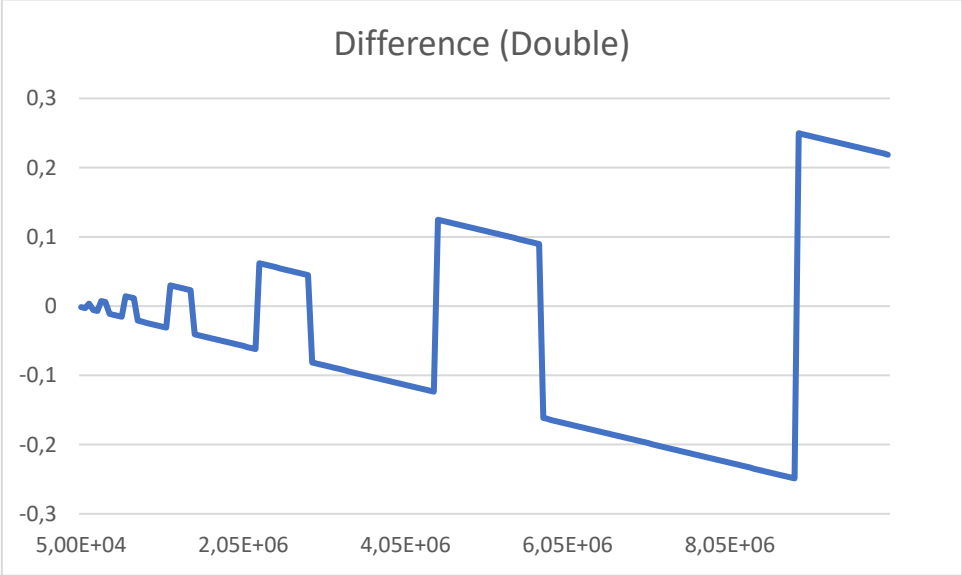
3.1. Kod wykorzystany w teście

```c
#define N 10000000
#define a 0.492710f
#define S 50000

int main(){
    printf("Calculating %f * %d (iteration)\n", a, N);

    double sum = 0;

    for(int i = 0; i*S < N; i++){
        for(int j = 0; j < S; j++)
            sum += a;

        double mul = (i+1)*S*a;

        printf("After %d:\tsum: %f\tmul: %f\tdiff: %f\terror: %.2f%%\n",
               (i+1)*S,
               sum, mul,
               mul - sum,
               abs(mul - sum) / mul * 100
        );
    }

    return 0;
}
```

3.2. Wyniki

Przy ilości sumowań $N = 10^7$ i kroku pomiarowym $S = 5 \cdot 10^4$ otrzymaliśmy po 200 dla pojedynczej i podwójnej precyzji, zbyt dużo aby je tu zamieszczać, są one dostępne w moim repozytorium.





3.3. Wnioski

Błąd nie jest stały, a co więcej jego znak może się zmieniać. W przypadku podwójnej precyzji widzimy pewien wzorzec, który jest spowodowany sposobem w jaki jest przechowywana liczba zmiennoprzecinkowa w systemie. W przypadku pojedynczej precyzji możemy podejrzewać że podobny wzorzec również występuje lecz jest nie widoczny w takiej skali (widać jedynie jego ślad przy małych wartościach).

Dla podwójnej precyzji różnica jest stosunkowo mała, dla 10^7 dodawań mieści się w ± 0.3. Nie można tego natomiast powiedzieć o operacjach na pojedynczej precyzji w przypadku której błąd sięgał prawie 10^6 co stanowi ponad 10% obliczanej wartości.

4. Alternatywne algorytmy dodawania

4.1. Algorytm rekurencyjny

Algorytm ten opiera się na dodawaniu rekurencyjnym (w postaci drzewa binarnego), co pozwala na zmniejszenie ilości dodawań oraz na dodawanie do siebie liczb o mniejszej różnicy cech. Sprawia to że algorytm ten jest dokładniejszy niż naiwne iteracyjne dodawanie, ale nadal może występować niedokładność.

4.1.1. Implementacja algorytmu

```
#define N 1000000000
#define a 0.492710f

float sum_rec(int n, float number){
    if(n == 0) return 0;
    if(n == 1) return number;
    return sum_rec(n/2, number) + sum_rec(n - n/2, number);
}

int main(){
    printf("Calculating %f * %d (recursive)\n", a, N);

    float sum = sum_rec(N, a);

    printf("Calculated as sum:\t%f\n", sum);
    printf("Calculated as mul:\t%f\n", N*a);
    printf("Difference:\t\t%f\n", abs(sum - N*a));
    printf("Error:\t\t\t%.2f%%\n", abs(sum - N*a)/(N*a)*100);

    return 0;
}
```

4.2. Algorytm sumacyjny Kahana

Algorytm ten znacząco (ale nie całkowicie) redukuje błędy powstałe przy dodawaniu liczb zmiennoprzecinkowych. Jest on podobny do naiwnego algorytmu iteracyjnego, tyle że dodatkowo dokonuje przeniesienia części, która normalnie była by utracona, do następnego dodawania.

4.2.1. Implementacja algorytmu

```
#define N 1000000000
#define a 0.492710f

int main(){
    printf("Calculating %f * %d (Kahan)\n", a, N);

    // algorytm Kahana
    float sum = a;
    float c = 0.0;
    for(int i = 1; i < N; i++){
        float y = a - c;
        float t = sum + y;
        c = (t - sum) - y;
        sum = t;
    }

    printf("Calculated as sum:\t%f\n", sum);
    printf("Calculated as mul:\t%f\n", N*a);
    printf("Difference:\t\t%f\n", abs(sum - N*a));
    printf("Error:\t\t\t%.2f%%\n", abs(sum - N*a)/(N*a)*100);

    return 0;
}
```

4.3. Porównanie algorytmów

4.3.1. Porównanie błędów

Różnica	10^7	10^8	10^9
Iteracyjny	70844.0	40882392.0	484321376.0
Rekurencyjny	0.0	0.0	32.0
Kahan	0.0	0.0	96.0

Powyższe pomiary wyraźnie pokazują że dodawanie iteracyjne jest naznaczone bardzo znacznym błędem (dla 10^9 jest to błąd rzędu 90%). Natomiast algorytmy rekurencyjny i Kahana można dla małych liczb uznać za bez błędne, błędy pojawiają się dopiero przy 10^9 ponieważ mamy do czynienia z przekroczeniem zakresu mantysy.

4.3.2. Porównanie czasu wykonania

Czas [s]	10^7	10^8	10^9
Iteracyjny	0.042	0.350	3.244
Rekurencyjny	0.091	0.832	8.037
Kahan	0.147	1.124	11.254

Pomiary pokazują że najszybszy jest algorytm iteracyjny, a najwolniejszy jest algorytm Kahana. Jest to o tyle ciekawe że porównując czasy do teoretycznych złożoności tych algorytmów widzimy jak duże znaczenie w przypadku rekurencyjnego ma konieczność wywoływania rekurencyjnego oraz w przypadku algorytmu Kahana konieczność dodatkowych operacji na liczbach zmiennoprzecinkowych.

4.4. Wnioski

Algorytm rekurencyjny i algorytm Kahana mają podobną dokładność, różnica polega na tym że rekurencyjny wymaga więcej pamięci, a Kahana więcej czasu. Zatem wybór pomiędzy nimi powinien być zależny od tego co jest dla nas ważniejsze, czas czy pamięć.