Metody Obliczeniowe w Nauce i Technice

Mateusz Kocot

Laboratorium 1 – raport

Zad. 1. Sumowanie liczb pojedynczej precyzji

1.1. Tablicę *arr* o $N = 10^7$ elementach wypełniono wartością $val = 0.7$:

```
vector<float> arr(N, val);
```

Następnie za pomocą funkcji *iterativeSum* obliczono sumę wszystkich N elementów tablicy *arr*.

```
float iterativeSum(vector<float> &arr)
{
    float sum = 0;
    for (auto el : arr)
    {
        sum += el;
    }
    return sum;
}
```

Otrzymano wynik: 6338540, podczas gdy prawdziwa suma wynosi 70000000.

2.2. Błędy obliczono w następujący sposób:

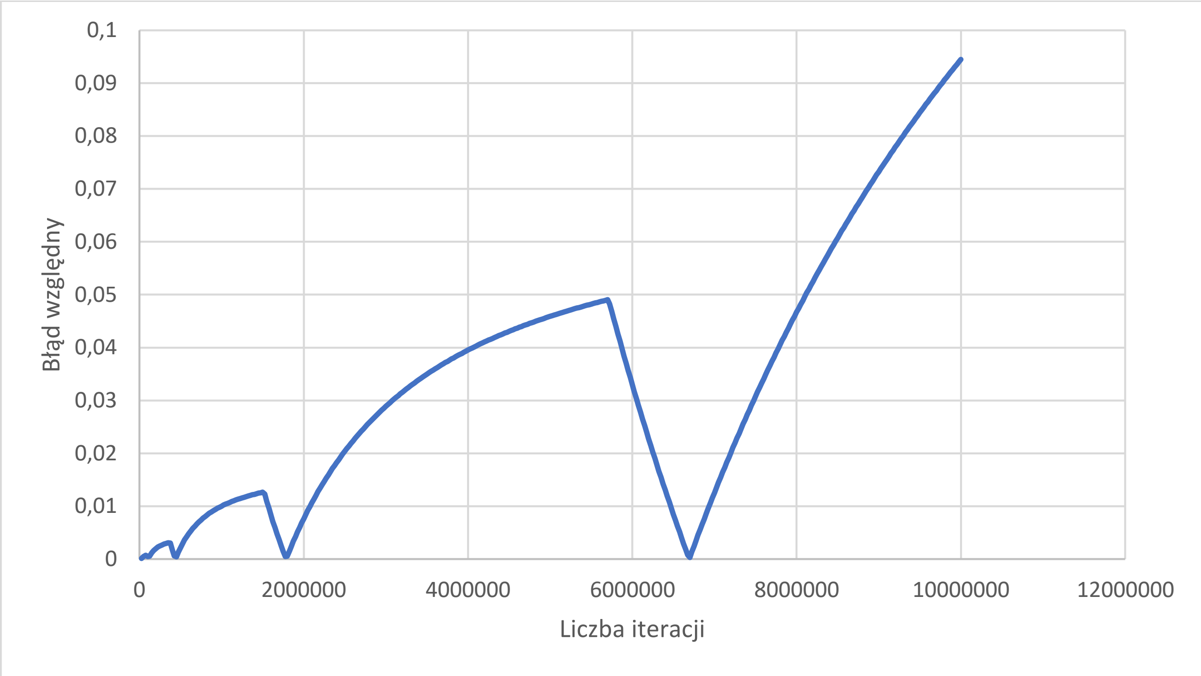
```
float absoluteError = abs(trueSum - arrSum);
float relativeError = absoluteError / trueSum;
```

Błędy wyniosły:

- Błąd bezwzględny: 661457
- Błąd względny: 0.0944939

Błąd względny jest tak duży, gdyż do dotychczasowej sumy cały czas dodawano tą samą liczbę. W związku z tym, po pewnej liczbie iteracji, dodawane były dwie liczby – jedna dużo większa od drugiej. Skutkowało to obcięciem mniej znaczących bitów.

2.3. Na poniższym wykresie przedstawiono zależność błędu względnego od ilości iteracji sumowania.



Wykres załamuje się w kilku miejscach. Po ich przekroczeniu, jeżeli wcześniej błąd bezwzględny rósł, zaczyna on maleć oraz na odwrót. Jest to spowodowane uwzględnianiem w sumowaniu nieco mniejszych lub większych liczb od $val = 0.7$. Mimo że wykres nie jest rosnący w każdym przedziale, można zauważyć, że wraz ze wzrostem iteracji sumowania, rośnie także maksymalna wartość błędu względnego, co jest spowodowane coraz większą różnicą między składnikami dodawania.

2.4. Zaimplementowano rekurencyjny algorytm *recursiveSum* sumujący zgodnie z zasadą „dziel i zwyciężaj”:

```
float recursiveSum(vector<float> &arr, int p, int r)
{
    if (p == r)
    {
        return arr[p];
    }
    int q = (p + r) / 2;
    return recursiveSum(arr, p, q) + recursiveSum(arr, q + 1, r);
}
```

2.5. Suma elementów wcześniej stworzonej tablicy obliczona z wykorzystaniem powyższej funkcji wyniosła tyle ile powinna, tzn. błąd bezwzględny wyniósł 0. W związku z tym, błąd względny także wyniósł 0. Błędy zmalały, gdyż algorytm dodaje do siebie liczby o tej samej wartości. Nie traci się więc informacji na mało znaczących bitach mantysy, tak jak ma to miejsce w przypadku dodawania do siebie liczb o sporej różnicy.

2.6. Wyznaczono czasy działania obu algorytmów dla tych samych danych wejściowych (wcześniej zdefiniowana tablica *arr*):

- *iterativeSum*: $0.082782\ s$,
- *recursiveSum*: $0.083778\ s$.

Różnica jest więc nieznaczna.

Zad. 2. Algorytm Kahana

Zaimplementowano algorytm Kahana:

```
float kahanSum(vector<float> &arr)
{
    float sum = 0;
    float err = 0;
    for (auto el : arr)
    {
        float y = el - err;
        float temp = sum + y;
        err = (temp - sum) - y;
        sum = temp;
    }
    return sum;
}
```

2.1. Podobnie, jak w przypadku algorytmu rekurencyjnego, błędy: względny i bezwzględny, w przypadku wykorzystania algorytmu Kahana do obliczenia sumy elementów tablicy z zad. 1., wynoszą 0.

2.2. Algorytm Kahana ma dużo lepsze własności dzięki braku utraty informacji na mało znaczących bitach (w stosunku do akumulatora *sum*). Do tego celu służy zmienna *err*.

2.3. Czas działania algorytmu Kahana dla tablicy *arr* wyniósł $0.101726\,s$. Algorytm ten działa więc zauważanie dłużej od algorytmu rekurencyjnego.

Zad. 3. Suma szeregu

Rozważany szereg:

$$\sum_{k=1}^{n} \frac{1}{2^{k+1}}$$

Dla $n = 50, 100, 200, 500, 800$.

3.1. Do sumowania w przód wykorzystano szablon funkcji:

```
template <class T>
T iterativeSum(vector<T> &arr)
{
    T sum = 0;
    for (auto el : arr)
    {
        sum += el;
    }
    return sum;

}
```

Do sumowania wstecz wykorzystano natomiast:

```
template <class T>
T iterativeSumBackward(vector<T> &arr)
{
    T sum = 0;
    for (int i = arr.size() - 1; i >= 0; i--)
    {
        sum += arr[i];
    }
    return sum;
}
```

Dla typu pojedynczej precyzji *float* suma szeregu dla każdego n wyniosła 0.5. Wynik byłby prawdziwy, gdyby $n \to \infty$. W tym przypadku widać niedoskonałość typu *float*, która sprawiła, że nawet dla $n = 50$, wyniki zostały zaokrąglone do 0.5 niezależnie od kolejności sumowania.

3.2. Analogiczny eksperyment przeprowadzono dla typu podwójnej precyzji *double*. Tutaj, niezależnie od kolejności sumowania, dla $n = 50$ uzyskano wynik: $0.49999999999999955591079014993738383054733276367187 5$. Dla pozostałych n, wyniki wyniosły 0.5.

3.3. Widać zatem, że typ *double* jest dokładniejszy od typu *float*. Dla $n = 50$ nie starczyło mantysy w typie *float* i wynik został zaokrąglony do jednego miejsca dziesiętnego. Dla typu *double* natomiast, zapełnione były aż 32 miejsca dziesiętne.

3.4. Do obliczenia sumy szeregu wykorzystano także algorytm Kahana. Zwracał on jednak takie same wyniki jak funkcje iteracyjne.

Zad. 4. Epsilon maszynowy

Napisano szablon funkcji służący do wyznaczenia epsilona maszynowego dla typu *float* oraz *double*:

```
template <class T>
T findEpsilon(T epsilon)
{
    T expression = 2;
    while (expression > 1)
    {
        epsilon /= 2;
        expression = 1 + epsilon;
    }
    return epsilon;
}
```

Wyniki:

- *float*: $5.96046 \cdot 10^{-8}$
- *double*: $1.11022 \cdot 10^{-16}$

Wyznaczone wartości są zgodne ze standardem IEEE 754.

Zad. 5. Algorytm niestabilny numerycznie

Przykładem algorytmu niestabilnego numerycznie może być liczenie metodą Eulera równania różniczkowego:

$$\frac{dy}{dx} = -y.$$

Rozwiązaniem analitycznym tego równania jest funkcja:

$$y = e^{-x}.$$

Jest to funkcja, która wraz ze wzrostem x do ∞, maleje do 0.

Rozwiążmy to równanie metodą Eulera (dla kroku h).

$$y_{i+1} = y_i + \frac{dy}{dx}h$$

$$y_{i+1} = y_i + (-y_i)h$$

$$y_{i+1} = y_i(1-h).$$

Ponieważ funkcja zmierza do 0 w ∞, mamy zależność:

$$\left|\frac{y_{i+1}}{y_i}\right| < 1.$$

Wynika stąd, że:

$$|1-h| < 1$$

$$h > 0 \wedge h < 2.$$

Wobec tego, gdy przyjmiemy $h \geq 2$, wartość bezwzględna y_i nie będzie maleć. Błąd między wartością rzeczywistą funkcji a wartością obliczoną algorytmem także nie będzie maleć, a dla $h > 2$ – będzie rosnąć. Zgodnie z definicją, będzie to wówczas algorytm niestabilny numerycznie.

Aby otrzymać wersję stabilną, wystarczy przyjąć $h \in (0, 2)$. Wówczas obliczane algorytmem wartości funkcji będą zmierzać do 0, a błąd będzie się zmniejszał.