# Laboratorium 1: Arytmetyka komputerowa

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

**Zadanie 1**

1.1 Utworzyłem wektor o wielkości 10^7 i wypełniłem go tą samą liczbą. Wybraną przeze mnie liczbą była 0.1634. Następnie obliczyłem sumę wszystkich liczb, które znajdują się w wektorze. Do wypełnienia wektora i obliczenia sumy stworzyłem specjalne funkcje.

```
float find_sum(std::vector<float> &numbers) {
      float result = 0.0f;
      for (float &number: numbers) {
      result += number;
      }
      return result;
}

void fill_vector_with_numbers(std::vector<float> &numbers) {
      for (float &number : numbers) {
      number = 0.1634f;
      }
}

//main()
std::vector<float> numbers;
numbers.resize(10000000);
fill_vector_with_numbers(numbers);
float sum = 0.0f;
sum = find_sum(numbers);
```

1.2 Za pomocą kalkulatora obliczyłem prawidłową sumę liczb i przypisałem ją do stałej. Obliczyłem błąd względny i bezwzględny wg wzorów:

- Błąd bezwzględny = |suma-otrzymana_suma|
- Błąd względny = |suma-otrzymana_suma|*100%/suma

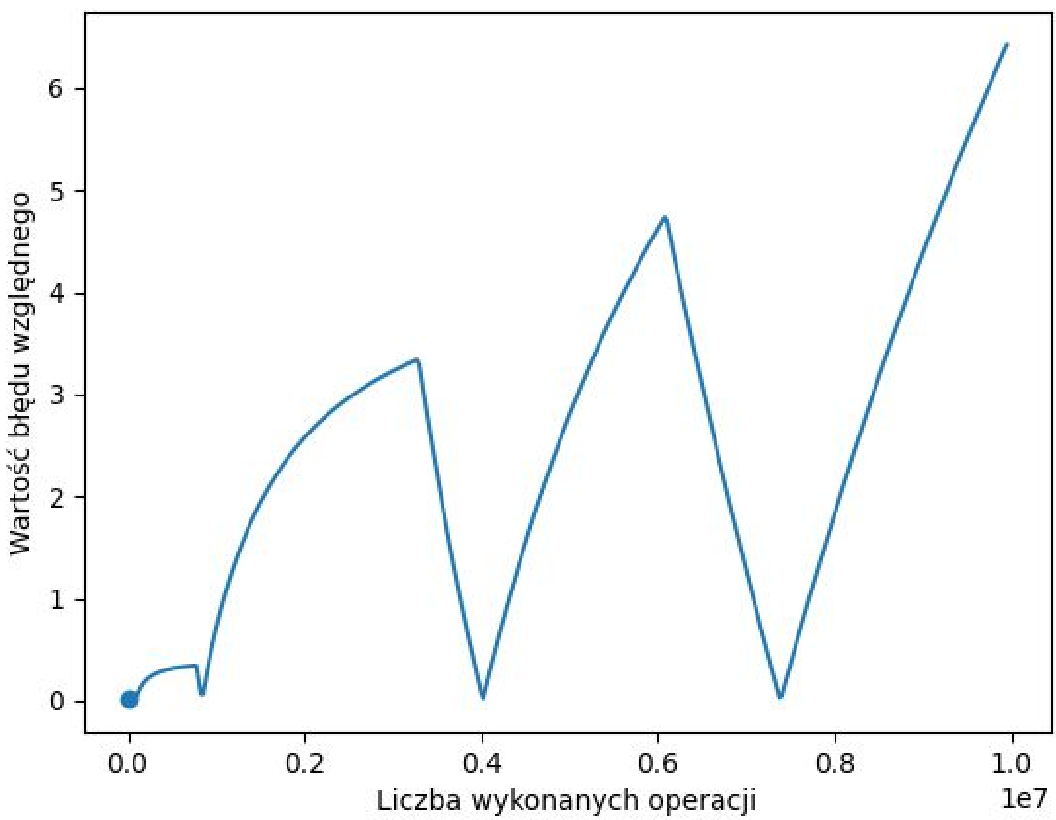
Dla liczby 0.1634 prawidłowa suma to 1634000, natomiast otrzymana to 1534553.750. Zatem błąd bezwzględny wynosi 99446.250, a błąd względny 6.086%.

Każda operacja na liczbach zmiennoprzecinkowych jest obarczona błędem. Błąd ten znacznie się zwiększa, gdy dodajemy dwie liczby różnych rzędów wielkości - tak jak ma to miejsce w tym przypadku.

1.3 Analiza zmiany błędu względnego.
W celu raportowania wielkości błędu względnego utworzyłem wektor, w którym będę zapisywał jego wartość co 25000 kroków.

```cpp
std::vector<float> add_errors;
sum = 0.0f;
for (int i = 0; i < numbers.size(); i++) {
    sum += numbers[i];
    if (i % 25000 == 0)
        add_errors.push_back(std::abs(i * 0.1634 - sum)*100/sum);
}
```

Wartości zmiany błędu względnego zapisałem w pliku wyniki.txt i na ich podstawie utworzyłem następujący wykres:




Wartość błędu względnego cyklicznie rośnie i maleje - zwiększając amplitudę w każdym cyklu. W wybranych punktach wartość błędu jest bliska 0%. Dla odpowiednio dobranej liczby operacji można uzyskać wynik zbliżony do rzeczywistego - pomimo ułomności algorytmu.

1.4-5 Rekurencyjny algorytm sumowania

Zaimplementowałem rekurencyjny algorytm sumowania liczb z wektora:

```
float find_sum_recursively(std::vector<float> &numbers, int p, int q) {
      if (p == q) return numbers[p];
      if (abs(p - q) == 1) return numbers[p] + numbers[q];
      int pivot = (p + q) / 2;
      return find_sum_recursively(numbers, p, pivot) +
             find_sum_recursively(numbers, pivot + 1, q);
}
```

Następnie obliczyłem błąd względny i bezwzględny sumy otrzymanej przy pomocy tego algorytmu:

- Błąd bezwzględny: 0.125
- Błąd względny: 0.00000765%

Błąd względny znacznie zmalał. Wynika to ze sposobu działania algorytmu. Algorytm rekurencyjny sumuje liczby parami - sumowane liczby mają ten sam lub nieznacznie różny rząd wielkości. Podczas sumowania w ten sposób nie tracimy zbyt dużo informacji.

1.6 Pomiar czasu działania algorytmów

Porównałem czas działania algorytmu sumującego liczby po kolei oraz algorytmu rekurencyjnego. Do zmierzenia czasu użyłem klasy **std::chrono::steady_clock.**

Wyniki pomiarów:

- Zwykłe sumowanie: **99ms**.
- Sumowanie rekurencyjne: **74ms**.

Algorytm rekurencyjny jest szybszy.

**Zadanie 2**

Korzystając z pseudokodu zaimplementowałem algorytm Kahana:

```
float kahan_sum(std::vector<float> &numbers){
      float sum = 0.0f;
      float err = 0.0f;
      for(float &number: numbers){
            float y = number - err;
            float tmp = sum + y;
            err = (tmp - sum) - y;
            sum = tmp;
      }
      return sum;
}
```

2.1 Błąd bezwzględny i względny
Wyliczyłem błędy dla tych samych danych wejściowych jak w przypadku zadania 1:

- Błąd bezwzględny: 0.000
- Błąd względny: 0.000%

2.2 Działanie algorytmu Kahana
Algorytm Kahana sprawdza na bieżąco jaki jest błąd dokładności sumowania.
Dokonuje tego przy pomocy zmiennej err - która przechowuje wartość aktualnego błędu sumowania(wartości na mniej znaczących bitach, które zostały utracone w wyniku zaokrąglenia). W następnych krokach wartość err jest odejmowana od liczby przed jej dodaniem do sumy, by wyrównać błąd będący wynikiem zaokrąglenia w poprzednim kroku.

2.3 Pomiar czasów działania algorytmów
Porównałem czas działania algorytmu sumującego liczby rekurencyjnie oraz algorytmu Kahana.
Do zmierzenia czasu użyłem klasy **std::chrono::steady_clock.**
Wyniki pomiarów:

- Sumowanie rekurencyjne: **74ms**.
- Algorytm Kahana: **125ms**

Algorytm rekurencyjny jest szybszy. Algorytm Kahana spowalnia wykonywanie dodatkowych operacji związanych ze sprawdzaniem błędu zaokrąglenia.

**Zadanie 3**
Zaimplementowany algorytm sumowania w przód dla pojedynczej precyzji:

```
float sum_forward(int n){
   float result = 0.0f;
   for(int i = 1; i <= n; i++){
       result += 1/(float)pow(2, i+1);
   }
   return result;
}
```

Ze względu na podobieństwo pozostałych trzech wersji algorytmu postanowiłem nie umieszczać ich w tym sprawozdaniu. Można je zobaczyć w pliku z3.cpp

3.1 Pojedyncza precyzja
Zaimplementowałem algorytm sumowania w przód i wstecz dla pojedynczej precyzji. Wyniki sumowania przedstawia poniższa tabela:

| | n = 50 | n = 100 | n = 200 | n = 500 | n = 800 |
|---|---|---|---|---|---|
| w przód | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| wstecz | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |

Wyniki są dokładnie takie same dla obu kolejności.

3.2 Podwójna precyzja
Eksperyment z punktu 3.1 powtórzyłem dla podwójnej precyzji. Otrzymałem następujące wyniki:

| | n = 50 | n = 100 | n = 200 | n = 500 | n = 800 |
|---|---|---|---|---|---|
| w przód | 0.4999999999999996 | 0.5 | 0.5 | 0.5 | 0.5 |
| wstecz | 0.4999999999999996 | 0.5 | 0.5 | 0.5 | 0.5 |

W dalszym stopniu otrzymane wyniki nie zależą od kolejności sumowania.

3.3 Porównanie wyników eksperymentu dla pojedynczej i podwójnej precyzji.
Wyniki różnią się tylko w jednym przypadku (dla n = 50). Różnica to wynika z ograniczonej precyzji typu float - wynik rzeczywisty został zaokrąglony w górę.

3.4 Sumowanie za pomocą algorytmu Kahana.
Po zastosowaniu algorytmu Kahana otrzymałem dokładnie takie same wyniki, jak przy prostym sumowaniu z podwójną precyzją.

**Zadanie 4**
Napisałem funkcję do wyznaczenia epsilona maszynowego:

```
double find_machine_epsilon(double eps = 1.0){
      double previous = eps;
      while((eps + 1.0) != 1.0){
            previous = eps;
            eps = eps/2;
      }
      return previous;
}
```

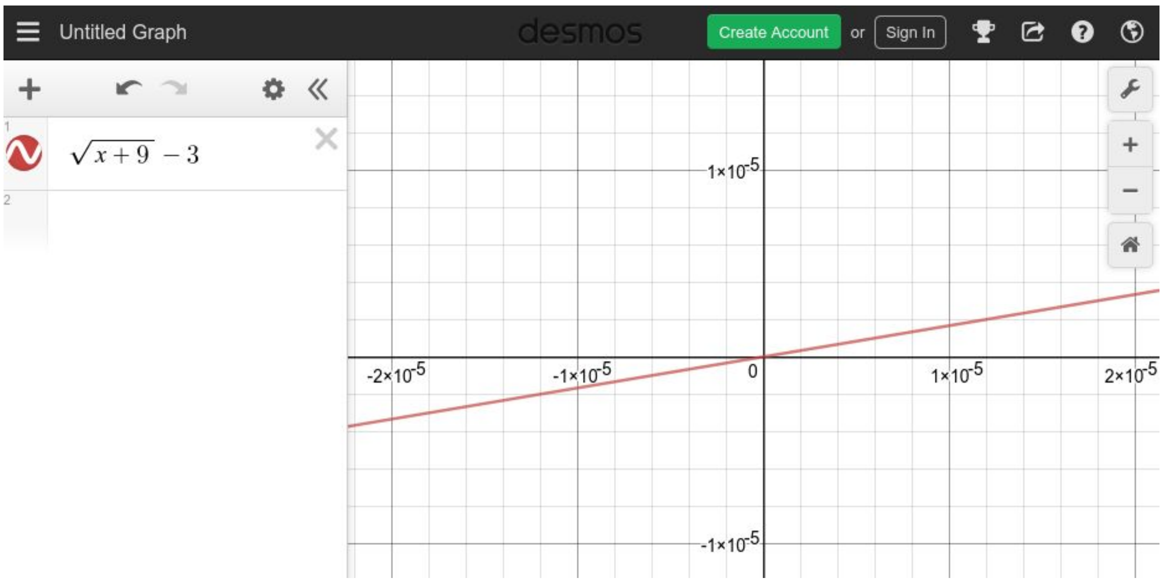
Funkcja zwraca wartość epsilona maszynowego równą: 0.000000000000000222 czyli 2.22e-16.
Według wikipedii(https://en.wikipedia.org/wiki/Machine_epsilon) otrzymałem wartość zgodną z rzeczywistością, co oznacza, że mój program działa poprawnie.

**Zadanie 5**
Algorytm, który wybrałem to program wyznaczający wartość funkcji:

$$\mathrm{f(x)} = \sqrt{x+9} - 3$$

w przedziale (-0.00001; 0.00001). Wyliczyłem wartości w 101 punktach różniących się o 0.0000002 na współrzędnej x.

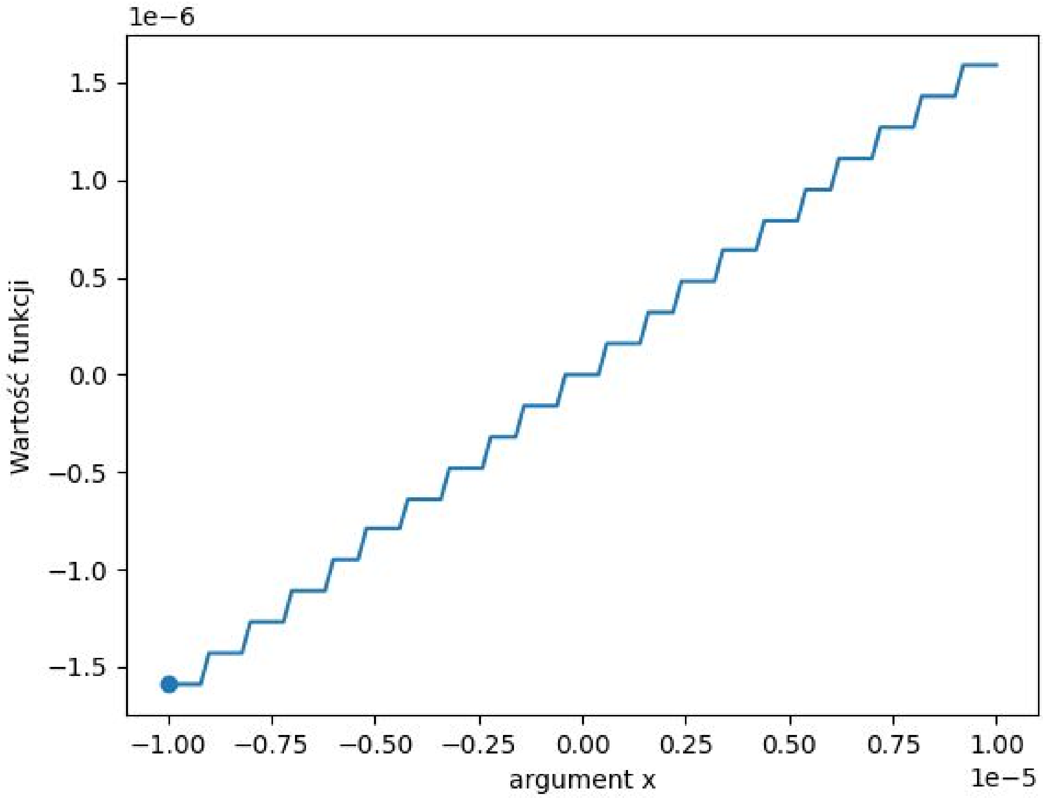
Wykres funkcji przypomina pół paraboli odwróconej o 90 stopni. Jednak na tak niewielkim przedziale, który badam w tym zadaniu fragment funkcji będzie przypominał zwykłą funkcję liniową. Poniżej zamieszczam zrzut ekranu ze strony desmos.com potwierdzający tę tezę.




5.1 Wersja niestabilna numerycznie
Jako wersję niestabilną numerycznie algorytmu przyjąłem po prostu funkcję wyznaczającą wartości funkcji f(x) w 101 punktach:

```
float unstable_version(float x){
    return (sqrt(x+9) - 3);
}
```

Następnie, wygenerowałem wykres wartości funkcji w tych punktach:




Otrzymany wykres nie przypomina funkcji liniowej. Wartości funkcji rosną skokowo - funkcja nie zmienia swojej wartości przy małym przyroście argumentu. Mamy więc do czynienia z algorytmem niestabilnym - otrzymany wynik jest widocznie przekłamany.

5.2 Utrata cyfr znaczących

We wzorze funkcji, dla argumentów x bliskich 0 liczba będąca wynikiem pierwiastka jest bliska 3. W przypadku obliczeń dokonanych przez komputer, tzn odjęcia od siebie dwóch bardzo zbliżonych wartości występuje **utrata cyfr znaczących.** To ona powoduje widoczną niedokładność algorytmu.

5.3 Wersja stabilna numerycznie

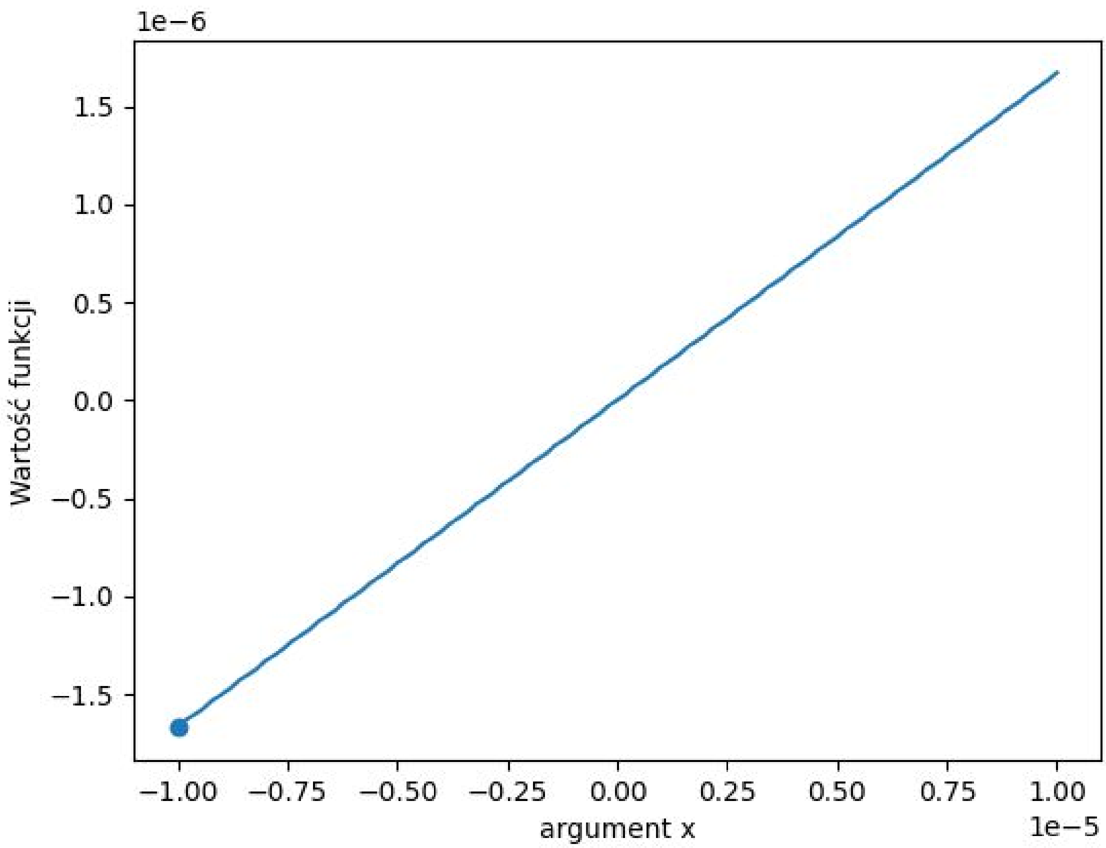
Korzystając ze wzorów skróconego mnożenia przekształciłem oraz zaimplementowałem funkcję:

$$\mathrm{f(x)} = \sqrt{x+9} - 3 = \frac{(\sqrt{x+9}-3)(\sqrt{x+9}+3)}{\sqrt{x+9}+3} = \frac{x}{\sqrt{x+9}+3}$$

```
float stable_version(float x){
    return x/(sqrt(x+9) + 3);
}
```

Przekształcony wzór funkcji nie zawiera operacji odejmowania, które powodowało utratę cyfr znaczących.

Wyznaczyłem wartości funkcji dla dokładnie tych samych danych wejściowych i utworzyłem wykres:




Wykres przypomina funkcję liniową, dlatego uważam, że utworzona wersja algorytmu jest stabilna.