AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody Obliczeniowe w Nauce i Technice:

Arytmetyka komputerowa

Laboratorium 1

Przemysław Lechowicz

SPIS TREŚCI

1. Sumowanie liczb pojedynczej precyzji

1.1. Napisz program, który oblicza sumę N liczb pojedynczej precyzji przechowywanych w tablicy o $N = 10^7$ elementach. Tablica wypełniona jest tą samą wartością v z przedziału [0.1, 0.9]. Wyznacz bezwzględny i względny błąd obliczeń. Dlaczego błąd względny jest tak duży?

```
int main()
{
	const auto value = 0.1f;
	const auto quantity = 10000000;
	float* tab = new float[quantity];
	auto sum = 0.0f;

	for (int i = 0; i < quantity; i++) {
		tab[i] = value;
	}

	for (int i = 0; i < quantity; i++) {
		sum += tab[i];
	}
```

Obliczona suma 10^7 pól tablicy o wartości $v = 0.1$ wyniosła

$$1.08794e + 06$$

Błąd bezwzględny wynosi:

$$|Zmierzona\ wartość\ -\ dokładna\ wartość| = |1.08794e06\ -\ 1e06|\ = 87937$$

Błąd względny wynosi:

$$\frac{Błąd\ bezwzględny}{dokładna\ wartość} = \frac{87937}{1e06} \cdot 100\% = 8.7937\%$$

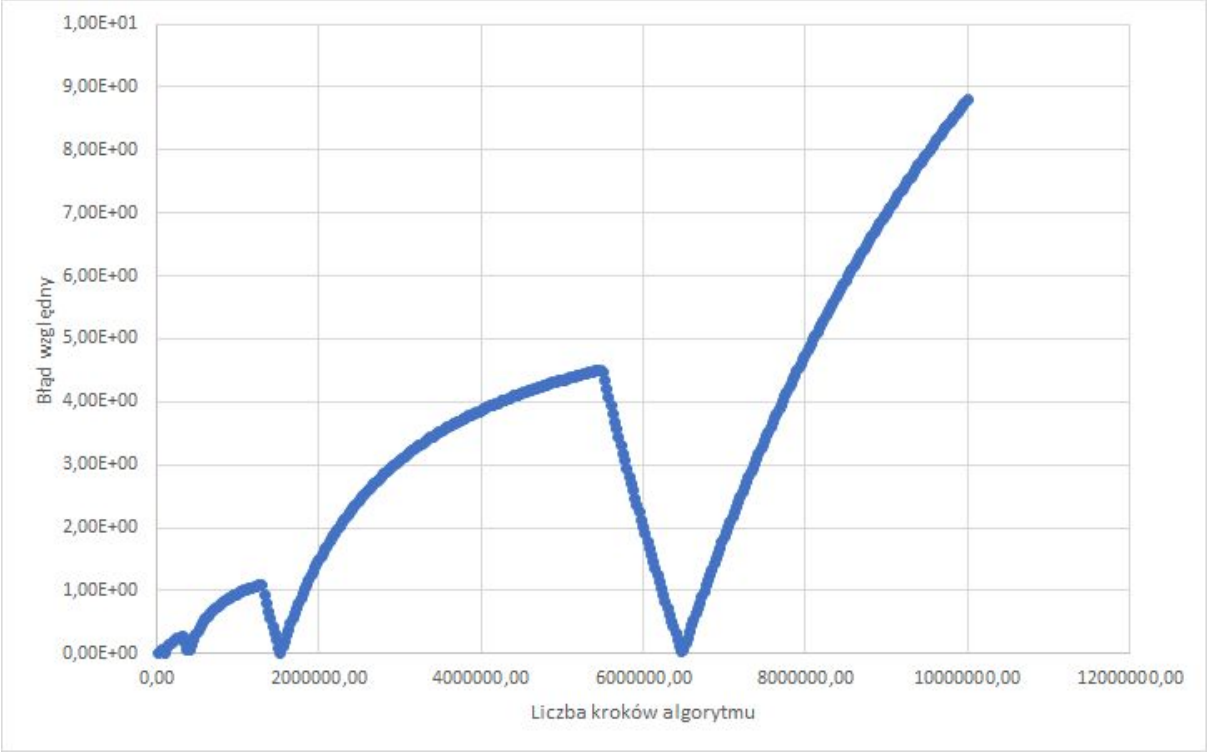
Liczba $0,1_{10}$ nie ma skończonego rozwinięcia w systemie binarnym - nie da się za pomocą komputera dokładnie pokazać tej liczby. Jest to przyczyną powstawania błędów bezwzględnych i względnych. Im więcej liczb do siebie dodamy tym większa będzie niedokładność oraz tym większy błąd otrzymamy.

1.2. W jaki sposób rośnie błąd względny w trakcie sumowania? Przedstaw wykres (raportuj wartość błędu co 25000 kroków) i dokonaj jego interpretacji.

Wyniki zostały zapisane do osobnego pliku results.txt.

```
std::fstream plik;
      plik.open("results.txt", std::ios::out);
      sum = 0;

      for (int i = 0; i < quantity; i++) {
            sum += tab[i];
            if ((i + 1) % 25000 == 0) {
                  float predicted = (i + 1) * value;
                  float bezwgl = abs(sum - predicted);
                  plik << (bezwgl / predicted) * 100 << std::endl;
            }
      }
      plik.close();
```



Jak możemy zauważyć, na wykresie występują “cykle”, którego poszczególne etapy są do siebie proporcjonalne. Im większy wzrost zauważamy, tym większy jest spadek.

Możemy zauważyć, że dla pewnych danych błąd względny spada i jest bardzo bliski 0%. Wnioskiem jest więc, że dla pewnej, nawet bardzo dużej liczby operacji możemy otrzymać bardzo mały błąd względny, a co za tym idzie - wynik bardzo zbliżony do rzeczywistości.

1.3. Zaimplementuj rekurencyjny algorytm sumowania wykorzystujący kolejne pary wartości. Wyznacz bezwzględny i względny błąd obliczeń. Dlaczego błąd względny znacznie zmalał?

```
float sum_rec(float tab[], int l, int r) {
      if (l == r) {
            return tab[l];
      }
      if (l < r) {
            int mid = (l + r) / 2;

            return sum_rec(tab, l, mid - 1) + tab[mid] +
sum_rec(tab, mid + 1, r);
      }
      return 0;
}
```

Obliczona suma 10^7 pól tablicy o wartości $v = 0.1$ wyniosła

$$1e + 06$$

Błąd bezwzględny wynosi:

$$0.125$$

Błąd względny wynosi:

$$1.25e - 05$$

1.4. Porównaj czas działania obu algorytmów dla tych samych danych wejściowych.

```
int normclock1 = clock();
for (int i = 0; i < quantity; i++) {
      sum += tab[i];
}
int normclock2 = clock() - normclock1;

int recclock1 = clock();
float recsum = sum_rec(tab, 0, quantity - 1);
int recclock2 = clock() - recclock1;
```

Czas działania dla algorytmu prostego wyniósł 0.029s.
Czas działania dla algorytmu rekurencyjnego wyniósł 0.251s.

Możemy zauważyć, że algorytm rekurencyjny jest ponad 8 razy wolniejszy od algorytmu prostego.

2. Algorytm Kahana

2.1. Wyznacz bezwzględny i względny błąd obliczeń dla tych samych danych wejściowych jak w przypadku testów z Zadania 1.

```
float KahanSum(float tab[], int quantity) {
      float sum = 0.0f;
      float err = 0.0f;
      for (int i = 0; i < quantity; i++) {
            float y = tab[i] - err;
            float temp = sum + y;
            err = (temp - sum) - y;
            sum = temp;
      }
      return (float)sum;
}
```

Obliczona suma 10^7 pól tablicy o wartości $v = 0.1$ wyniosła

$$1e+06 .$$

Błąd bezwzględny wynosi:

$$0.0$$

Błąd względny wynosi:

$$0.0\ \%$$

2.2. Wyjaśnij dlaczego algorytm Kahana ma znacznie lepsze własności numeryczne? Do czego służy zmienna err?

Algorytm Kahana znacznie zmniejsza błąd liczbowy w porównaniu z oczywistym podejściem. Odbywa się to poprzez zachowanie osobnej zmiennej err służącej do kumulacji małych błędów.

2.3. Porównaj czasy działania algorytmu Kahana oraz algorytmu sumowania rekurencyjnego dla tych samych danych wejściowych.

```
int kahanclock1 = clock();
      float kahan = KahanSum(tab, quantity);
      int kahanclock2 = clock() - kahanclock1;
```

Czas działania dla algorytmu rekurencyjnego wyniósł 0.251s.
Czas działania dla algorytmu Kahana wyniósł 0.113s.

Możemy zauważyć, że algorytm Kahana jest ponad 2 razy szybszy od algorytmu rekurencyjnego.

3. Suma szeregu

3.1. Oblicz wartość szeregu dla pojedynczej precyzji. Dokonaj sumowania zarówno w przód jak i wstecz. Porównaj wyniki dla obu kolejności

```
float seriesSum(int N) {
	float sum = 0.0;

	for (int k = 1; k <= N; k++) {
		sum += (float)1 / (float)pow(2, k + 1);
	}
	return sum;
}

float seriesSumBack(int N) {
	float sum = 0.0;

	for (int k = N; k >= 1; k--) {
		sum += (float)1 / (float)pow(2, k + 1);
	}
	return sum;
}
```

Mimo ustawienia wysokiej precyzji dla każdej wartości n i niezależnie od sumowania w przód czy w tył wynik zawsze był jednakowy i wynosił 0.5.

3.2. Powtórz analogiczny eksperyment dla podwójnej precyzji. Dokonaj porównaniu wyników, które otrzymałeś w dwóch poprzednich punktach.

Podobnie jak w przypadku pojedynczej precyzji dla wszystkich przypadków wynik wyniósł 0.5, z wyjątkiem sytuacji, gdy n = 50. Wtedy niezależnie od sumowania w przód czy w tył wynik wyniósł 0.49999999999999956.

Po wynikach można wywnioskować, że zmienne podwójnej precyzji są dokładniejsze. Dokładny wynik takiej sumy dla n = 50 wynosi 0.4999999999999995559, więc program przybliża dokładnie do 16 miejsca po przecinku. Dla wyższych n kolejne liczby po przecinku wynoszą 9 i program zaokrągla całą liczbę do 0.5. Dla zmiennych pojedynczej precyzji następuje to dla mniejszej liczby miejsc po przecinku.

3.3. Do obliczenia szeregu wykorzystaj algorytm Kahana z 2 zadania i sprawdź czy obliczane wartości oraz błąd ulegną poprawie.

Algorytm Kahana również podał wynik 0.5 dla wszystkich n, oprócz n=50. Dla tego n podał wynik 0.49999999999999978, co jest wynikiem dalszym prawdy niż proste sumowanie.

4. Epsilon maszynowy

4.1. Napisz program w języku C/C++ służący do wyznaczania epsilona maszynowego. Sprawdź, czy wynik jest zgodny ze standardem IEEE 754. Jeśli nie, to dlaczego? Jak to naprawić?

```
float floatEpsilon = 1.0f;

while (1 + floatEpsilon > 1) {
      floatEpsilon /= 2;
}
std::cout << 2 * floatEpsilon << std::endl;
```

Stosując prosty algorytm wyliczania epsilonu maszynowego otrzymane zostały wyniki:

- 1.19209e-07 dla zmiennej pojedynczej precyzji,
- 2.22045e-16 dla zmiennej podwójnej precyzji.

Jest to wynik zgodny ze standardem IEEE 754 (Według prof. Highama; Norma ISO C; Stałe języka C, C ++ i Python; Mathematica, MATLAB i Octave; różne podręczniki),

5. Algorytm niestabilny numerycznie

5.1. Wymyśl własny przykład algorytmu niestabilnego numerycznie, a następnie:

5.1.1. W oparciu o definicję algorytmu niestabilnego zademonstruj, że działa niepoprawnie. Wyjaśnij dlaczego.

```
void unstable() {
	for (float value = -1.0; value <= 1.0; value += 0.2) {
		std::cout << value << std::endl;
	}
}
```

Zaproponowany przeze mnie algorytm niestabilny numerycznie to algorytm obliczania w pętli liczb różniących się od siebie o 0.2, w zakresie od -1 do 1.
Algorytm zamiast oczekiwanej liczby 0 jako piąta, kolejna liczba pokazuje liczbę -3.27826e-08, poprzez niedokładność typu float.
Nie jest to algorytm stabilny, ponieważ dopuścił do sytuacji, w której otrzymaliśmy przekłamany wynik, poprzez brak liczb znaczących.

5.1.2. Wprowadź modyfikacje, które pozwolą otrzymać wersję stabilną.

```
void stable() {
	for (int value = -10; value <= 10; value += 2) {
		std::cout << (float)value/10 << std::endl;
	}
}
```

Liczby niecałkowite możemy obliczać za pomocą funkcji, która przyjmuje wartości 10-krotnie razy większe, a na wyjściu przeprowadza konwersję do typu zmiennoprzecinkowego i dzieli je przez 10.