AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

# DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

## Obliczanie liczby PI z podziałem na wątki

Autor:
Łukasz Nowak

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

# Spis treści

**1. Ogólne określenie wymagań** **3**

**2. Analiza problemu** **4**

2.1. Zastosowanie algorytmu . . . . . 4

2.2. Sposób działania programu . . . . . 4

**3. Projektowanie** **6**

3.1. Użyte narzędzia . . . . . 6

3.2. Diagram klas (uproszczony) . . . . . 6

3.3. Schemat działania algorytmu . . . . . 7

3.4. Uwagi dodatkowe . . . . . 7

**4. Implementacja** **8**

4.1. Funkcja calculate_integral . . . . . 8

4.2. Funkcja thread . . . . . 9

4.3. Zawartość funkcji main . . . . . 9

4.4. Tabela zawierająca wyniki pomiarów . . . . . 11

**5. Przedstawienie wyników za pomocą wykresu** **14**

**6. Wnioski** **16**

6.1. Opis współpracy . . . . . 16

6.2. Możliwe pytania . . . . . 17

6.3. Podsumowanie . . . . . 18

**Literatura** **19**

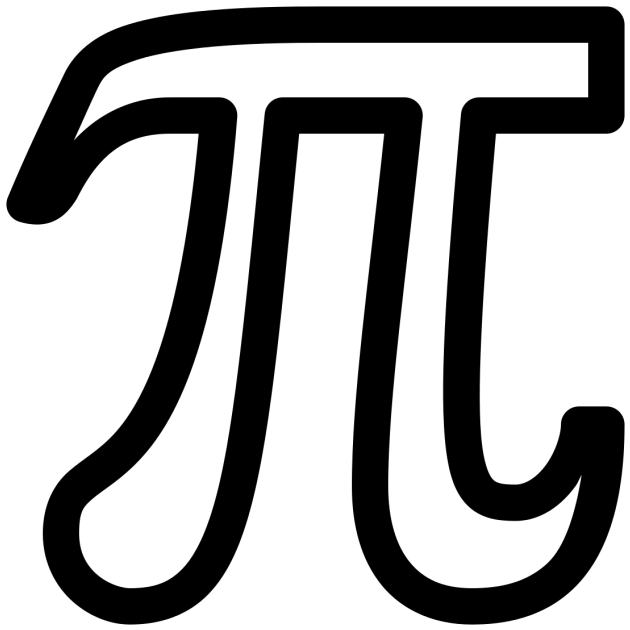
**Spis rysunków** **20**

**Spis tabel** **21**

**Spis listingów** **22**

# 1. Ogólne określenie wymagań

Program w języku C++ ma na celu obliczenie przybliżonej wartości liczby $\pi$, korzystając z metody całkowania numerycznego. Użytkownik ma możliwość dostosowania ilości punktów z przedziału całkowania oraz ilości wątków, co umożliwia zrównoleglenie obliczeń. Implementacja zrównoleglenia opiera się na bibliotece thread zgodnej ze standardem POSIX. Program prezentuje na ekranie terminala czas trwania obliczeń oraz uzyskaną wartość $\pi$. Obrazek znaleźć można pod tym adresem [1].



**Rys. 1.1.** Liczba PI

Projekt ma zostać napisany przy pomocy sztucznej inteligencji o nazwie ChatGPT, z którego można kożystać pod adresem [2]. Współpraca ma za zadanie wyznaczenia czynności do wykonania przez bota, testowanie poprawności kodu oraz wyciągnięcie wniosków ze współpracy i przygotowanie dokumentacji podsumowującej całe zadanie.

# 2. Analiza problemu

## 2.1. Zastosowanie algorytmu

Algorytm ten jest używany do przybliżonego obliczania liczby $\pi$ przy użyciu metody Monte Carlo. Metoda Monte Carlo wykorzystuje losowe próbki, aby estymować wyniki numeryczne, w tym wartości liczbowe. W przypadku tego algorytmu, używa się go do obliczania wartości $\pi$ na podstawie całki numerycznej.

## 2.2. Sposób działania programu

1. Program przyjmuje od użytkownika ilość wątków, precyzję wyniku oraz określony przedział całkowania [a, b].
2. Następnie program tworzy określoną liczbę wątków, z których każdy oblicza swoją część całki używając metody Monte Carlo.
3. Wyniki z poszczególnych wątków są sumowane, aby uzyskać całkowity wynik całki.
4. Ostateczny wynik jest wyświetlany z określoną precyzją, a także podawany jest czas, jaki upłynął od rozpoczęcia do zakończenia obliczeń.

**Metoda Monte Carlo**: Metoda Monte Carlo to rodzaj numerycznej techniki obliczeniowej, która opiera się na losowaniu próbek, aby uzyskać przybliżoną wartość pewnej wielkości matematycznej. Nazwa tej metody pochodzi od kasyna Monte Carlo w Monako, gdzie hazard jest grą opartą na czystym szczęściu.
**Przykład zastosowania: Obliczanie liczby $\pi$**:

Definicja problemu:

Chcemy obliczyć wartość liczby $\pi$, czyli stosunek obwodu koła do jego średnicy.
Generowanie próbek:

Losujemy punkty (x, y) w kwadracie o boku długości 1. Te punkty są naszymi próbkami.
Obliczenia na podstawie próbek:

Dla każdego punktu sprawdzamy, czy leży on wewnątrz koła o promieniu 0.5 (czyli kwadracie o środku w punkcie (0.5, 0.5) i boku długości 1). Jeśli tak, to zliczamy ten punkt jako punkt wewnątrz koła.
Średnia wartość:

Stosunek liczby punktów wewnątrz koła do wszystkich punktów daje nam przy-

bliżoną wartość stosunku obszaru koła do obszaru kwadratu, czyli przybliżoną wartość liczby $\pi/4$.
Oszacowanie błędu:

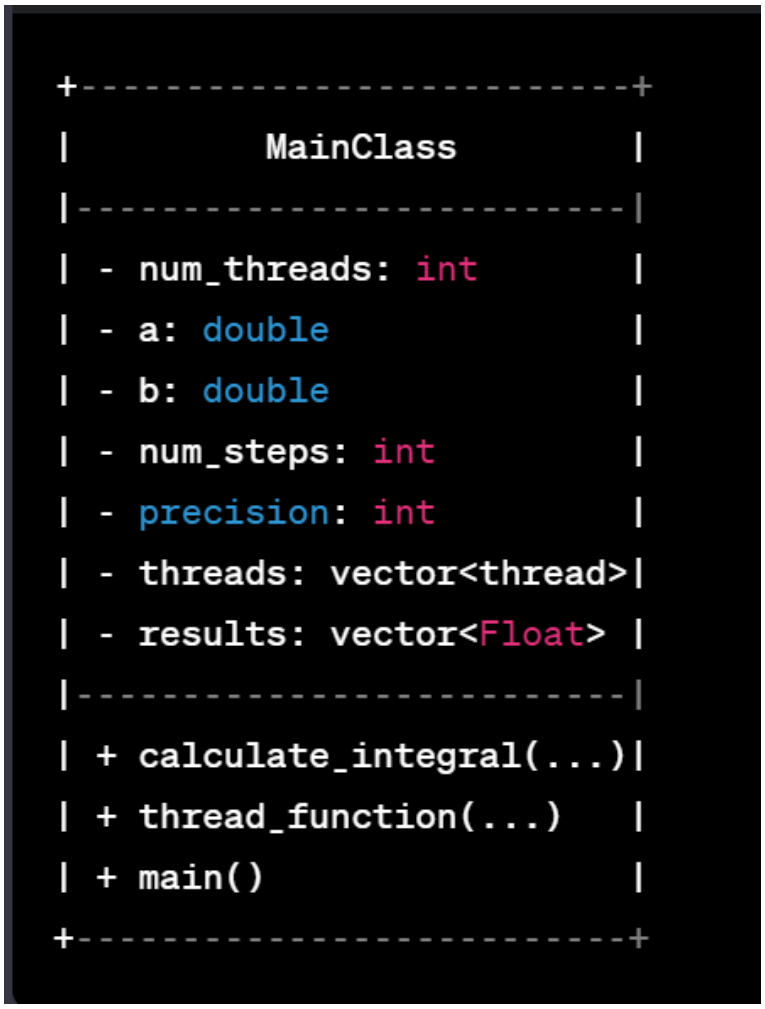
Oszacowujemy błąd na podstawie ilości punktów i stosunku punktów wewnątrz koła do wszystkich punktów. Metoda Monte Carlo jest bardzo elastyczna i może być używana do rozwiązywania różnych problemów matematycznych, zwłaszcza tam, gdzie tradycyjne metody analityczne są trudne do zastosowania.

# 3. Projektowanie

## 3.1. Użyte narzędzia

**Język programowania**: C++
**Kompilator**: g++
**Biblioteki**: iostream, iomanip, cmath, thread, vector, chrono

## 3.2. Diagram klas (uproszczony)




**Rys. 3.1.** Wygenerowany przez bota diagram klas

Na rysunku 3.1 został przedstawiony diagram klas stworzony na podstawie kodu.

## 3.3. Schemat działania algorytmu

1. Użytkownik wprowadza ilość wątków (num_threads), przedział całkowania (a i b), ilość kroków całkowania (num_steps), oraz precyzję wyniku (precision).
2. Tworzymy wektor wątków (threads) oraz wektor wyników z poszczególnych wątków (results).
3. W funkcji calculate_integral obliczamy wartość całki przy użyciu metody Monte Carlo.
4. W funkcji thread_function każdy wątek oblicza swoją część całki. 5. W funkcji main tworzymy i uruchamiamy wątki, a następnie czekamy na ich zakończenie. 6. Sumujemy wyniki z poszczególnych wątków. 7. Wyświetlamy wynik z określoną precyzją oraz czas obliczeń.

## 3.4. Uwagi dodatkowe

Program używa podwójnej precyzji (FloatType = double), co pozwala uzyskać dokładniejsze wyniki.
Ilość kroków całkowania (num_steps) została ustalona na dużą liczbę, co zwiększa dokładność wyniku, ale również wydłuża czas obliczeń.
Program korzysta z biblioteki ¡chrono¿ do pomiaru czasu wykonania obliczeń.
Możliwość wczytywania ilości miejsc po przecinku oraz innych parametrów została zakomentowana w celu uproszczenia dla przykładu, ale może być odkomentowana w praktycznych zastosowaniach.
W kodzie użyto wielowątkowości (std::thread) w celu równoległego obliczania wartości całki.

# 4. Implementacja

## 4.1. Funkcja calculate_integral

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <thread>
#include <vector>
#include <chrono>
int num_threads;
using FloatType = double; // Uzycie podwojnej precyzji

FloatType calculate_integral(double start, double end, int
    num_steps) {
    FloatType step = (end - start) / num_steps;
    FloatType sum = 0.0;

    for (int i = 0; i < num_steps; ++i) {
        FloatType x = start + (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    return step * sum;
}
```

**Listing 1.** Funkcja calculate_integral

Na dostarczonym listingu o numerze 1 zostały podane biblioteki wykorzystywane przez utworzony algorytm oraz funkcja calculate_integral.
Funkcja ta jest odpowiedzialna za obliczanie przybliżonej wartości całki za pomocą metody prostokątów (metody prostokątów z punktem środkowym). Jest to często stosowana metoda w numerycznych obliczeniach całkowych.

1. start i end reprezentują granice przedziału całkowania, a num_steps to liczba kroków, na jakie dzielony jest ten przedział.
2. step to długość kroku całkowania, obliczana na podstawie granic przedziału i liczby kroków. 3. W pętli for iterujemy przez wszystkie kroki całkowania.
4. Dla każdego kroku obliczamy wartość zmiennej x jako środek przedziału.
5. Dodajemy do sumy wartość funkcji podcałkowej dla danego x. W tym przypadku, używamy wzoru funkcji f(x) = 4 / (1 + x$\hat{2}$).
6. Ostateczna wartość całki to suma wartości funkcji pomnożona przez długość kroku (step).

## 4.2. Funkcja thread

```
void thread_function(int thread_id, double a, double b, int
    num_steps, int thread_count, std::vector<FloatType>& results) {
     results[thread_id] = calculate_integral(a + thread_id * (b - a)
     / thread_count, a + (thread_id + 1) * (b - a) / thread_count,
    num_steps / thread_count);
}
```

**Listing 2.** Funkcja thread

Funkcja thread_function, przedstawiona na listingu 2, jest funkcją wykonywaną przez każdy wątek w programie. Jest to część kodu, który odpowiada za obliczenia częściowe całki na określonym fragmencie przedziału całkowania dla danego wątku.

1. thread_id: Numer identyfikacyjny danego wątku, używany do określenia, którą część przedziału ma obliczyć.
2. a i b: Granice przedziału całkowania.
3. num_steps: Ilość kroków całkowania.
4. thread_count: Całkowita liczba wątków biorących udział w obliczeniach.
5. results: Wektor, w którym przechowywane będą wyniki obliczeń poszczególnych wątków.

## 4.3. Zawartość funkcji main

```
int main() {
    const double a = 0.0;  // Poczatek przedzialu calkowania
    const double b = 1.0;  // Koniec przedzialu calkowania
    const unsigned long long int num_steps = 440000000000;  //
    Ilosc krokow calkowania

    int precision = 60;  // Ilosc miejsc po przecinku

    /* Wczytywanie ilosci miejsc po przecinku od uzytkownika
    std::cout << "Podaj ilosc miejsc po przecinku: ";
    std::cin >> precision;
    */

    // Wczytywanie ilosci watkow od uzytkownika
    std::cout << "Podaj ilosc watkow: ";
    std::cin >> num_threads;

    std::vector<std::thread> threads;
```

```
18      std::vector<FloatType> results(num_threads); // Przechowywanie
    wynikow z poszczegolnych watkow
19
20      auto start_time = std::chrono::high_resolution_clock::now();
21
22      // Tworzenie i uruchamianie watkow
23      for (int i = 0; i < num_threads; ++i) {
24          threads.emplace_back(thread_function, i, a, b, num_steps,
    num_threads, std::ref(results));
25      }
26
27      // Oczekiwanie na zakonczenie wszystkich watkow
28      for (auto& thread : threads) {
29          thread.join();
30      }
31
32      FloatType result = 0.0;
33
34      // Sumowanie wynikow z wektora
35      for (auto& result_thread : results) {
36          result += result_thread;
37      }
38
39      auto end_time = std::chrono::high_resolution_clock::now();
40      auto duration = std::chrono::duration_cast<std::chrono::
    microseconds>(end_time - start_time);
```

**Listing 3.** Funkcja main

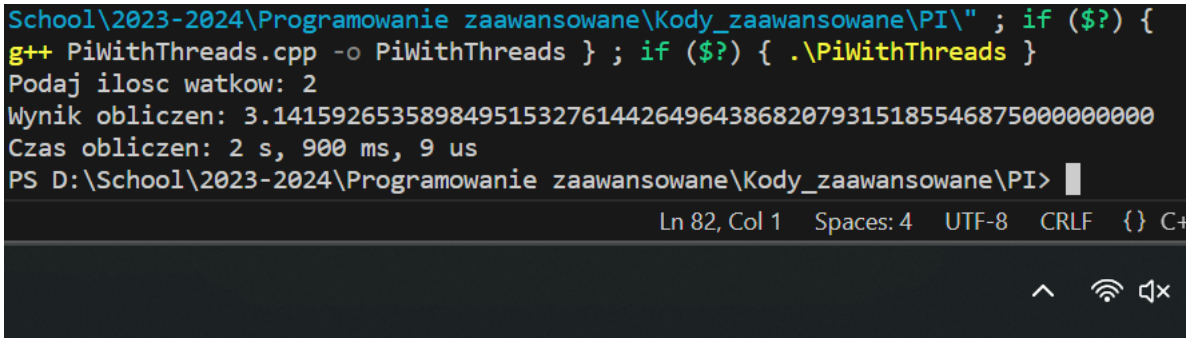
Na listingu 3 została przedstawiona część funkcji main odpowiadająca za backend programu. Zostaje tutaj ustawiona dokładność liczenia liczby $\pi$ w linii 4-tej listingu oraz ilość wyświetlanych miejsc po przecinku w linii 6-tej. Następnie program pyta użytkownika o ilość wątków na jakie ma zostać podzielone liczenie liczby $\pi$ i wpisanie tej liczby do zmiennej num_threads. W momencie wciśnięcia ENTER zostaje uruchomiony licznik czasu i program rozpoczyna liczenie. Czynność ta jest przedstawiona na rysunku 4.1.

Wyniki do tych obliczeń przedstawione na rysunku 4.2. Rysunek przedstawia wynik podanych obliczeń z wcześniej ustaloną liczbą miejsc po przecinku oraz z dużą dokładnością 440 000 000 000 próbek.

Kod odpowiedzialny za te wyniki znajduje się na listingu 4. Czas obliczeń jest podawany w sekundach, milisekundach oraz mikrosekundach z odpowiednimi oznaczeniami.

```
1
```


**Rys. 4.1.** Początek wykonywania obliczeń




**Rys. 4.2.** Zakończenie obliczeń oraz wyświetlenie wyników z czasem obliczeń

```
2  // Wyswietlanie wyniku z okreslona iloscia miejsc po przecinku
3      std::cout << std::fixed << std::setprecision(precision);
4      std::cout << "Wynik obliczen: " << result << std::endl;
5
6      // Wyswietlanie czasu obliczen w sekundach, milisekundach i
       mikrosekundach
7      std::cout << "Czas obliczen: "
8                << std::chrono::duration_cast<std::chrono::seconds>(
       duration).count() << " s, "
9                << std::chrono::duration_cast<std::chrono::
       milliseconds>(duration).count() % 1000 << " ms, "
10               << duration.count() % 1000 << " us" << std::endl;
11
12     return 0;
13 }
```

**Listing 4.** Funkcja main - kontynuacja

## 4.4. Tabela zawierająca wyniki pomiarów

Tabela przedstawiająca dane czasowe względem ilości wątków została przedstawiona pod numerem 4.1 oraz jej dalsza część w tabeli o numerze 4.2.

| Zestawienie czasu do liczby wątków | | | |
|---|---|---|---|
| Liczba wątków | Czas pomiaru (ms) | | |
| | Liczba Podziału: 100 mln | Liczba Podziału: 1 mld | Liczba Podziału: 3 mld |
| 1 | 294,9 | 2951,8 | 8577 |
| 2 | 156,3 | 1488 | 4381,3 |
| 3 | 101,7 | 1026,3 | 3000,5 |
| 4 | 83,4 | 789 | 2312,5 |
| 5 | 66,3 | 632,3 | 1905,8 |
| 6 | 61,1 | 552 | 1619,9 |
| 7 | 51,8 | 475,6 | 1480,4 |
| 8 | 46,9 | 442 | 1370,3 |
| 9 | 42,7 | 398,2 | 1221,3 |
| 10 | 38,8 | 367,1 | 1119,3 |
| 11 | 35,7 | 339,1 | 1052,2 |
| 12 | 33,9 | 343 | 1032,7 |
| 13 | 51,5 | 370,9 | 1072,1 |
| 14 | 50 | 345,3 | 1090,7 |
| 15 | 50 | 364,8 | 1037,6 |
| 16 | 44 | 391,5 | 1070,1 |
| 17 | 44 | 403,9 | 1065,4 |
| 18 | 42 | 412,8 | 1029,3 |
| 19 | 42 | 394,8 | 1068,6 |
| 20 | 40 | 391,6 | 1061,2 |
| 21 | 39,6 | 398,1 | 1102,1 |
| 22 | 36,9 | 350,2 | 1106 |
| 23 | 37,2 | 356,7 | 1115,3 |
| 24 | 37,8 | 385,4 | 1087,3 |
| 25 | 43,1 | 348,4 | 1063,9 |
| 26 | 43,4 | 355,8 | 1058,3 |
| 27 | 41,8 | 340,2 | 1101 |
| 28 | 40 | 362,5 | 1079,7 |
| 29 | 39 | 346,8 | 1039,4 |
| 30 | 38,9 | 356,4 | 1035,2 |

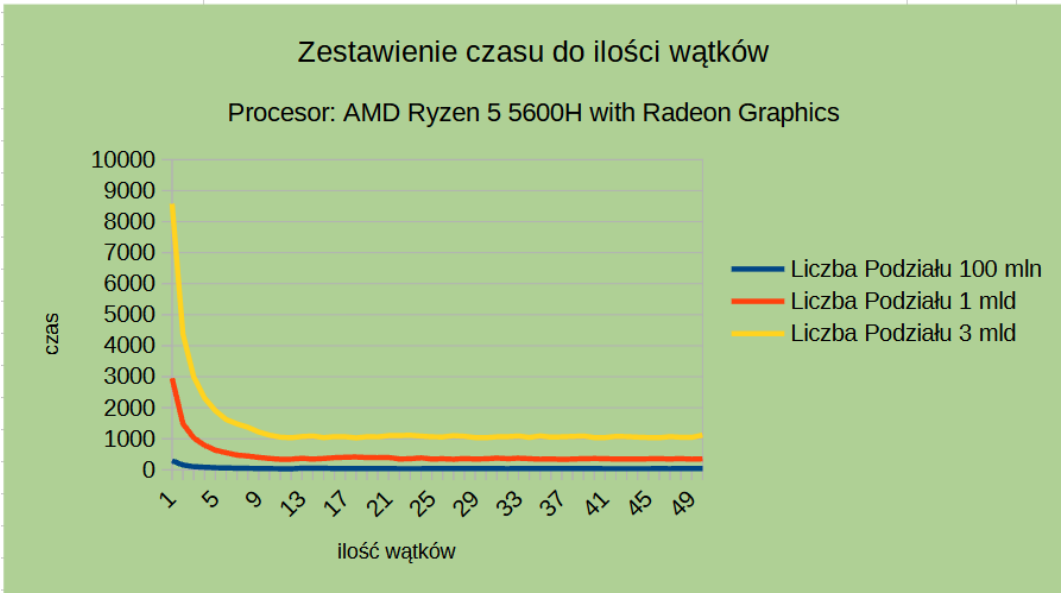
**Tab. 4.1.** Zestawienie czasów pomiaru z liczbami wątków

| Zestawienie czasu do liczby wątków | | | |
|---|---|---|---|
| Liczba wątków | Czas pomiaru (ms) | | |
| | Liczba Podziału: 100 mln | Liczba Podziału: 1 mld | Liczba Podziału: 3 mld |
| 1 | 38,7 | 377,8 | 1063,1 |
| 2 | 37,8 | 359 | 1067,8 |
| 3 | 42,4 | 375,7 | 1091 |
| 4 | 43,9 | 358,7 | 1042,8 |
| 5 | 41,3 | 343,9 | 1093,9 |
| 6 | 41,6 | 352,3 | 1052,2 |
| 7 | 41,3 | 335,2 | 1062,4 |
| 8 | 40,8 | 347,2 | 1075 |
| 9 | 39,4 | 358 | 1090,8 |
| 10 | 39 | 364,5 | 1040,9 |
| 11 | 38 | 357,2 | 1040,7 |
| 12 | 37,9 | 347,5 | 1079,1 |
| 13 | 37,5 | 351,3 | 1070,3 |
| 14 | 38 | 344,5 | 1053,4 |
| 15 | 38,1 | 354,2 | 1040,1 |
| 16 | 39,2 | 360,6 | 1032,8 |
| 17 | 37,6 | 346,8 | 1069,5 |
| 18 | 40,7 | 359,4 | 1048,8 |
| 19 | 39,5 | 342,2 | 1045,2 |
| 20 | 38,4 | 353,1 | 1117,9 |

**Tab. 4.2.** Zestawienie czasów pomiaru z liczbami wątków

# 5. Przedstawienie wyników za pomocą wykresu

Wyniki z tabeli przedstawione zostały na wykresie oznaczonym numerem 5.1. Należy zauważyć że wyniki jakie można otrzymać będą się różnić w zależności od procesora, algorytmu, kompilatora czy też metody użytej podczas liczenia liczby $\pi$.




**Rys. 5.1.** Wyniki na wykresie

W moim przypadku procesor, którego używam, to AMD Ryzen 5 5600H. Obliczanie liczby $\pi$ nie sprawiało mu większych problemów przy mniejszej ilości próbek ale w okolicy 2 mld liczba na jednym wątku zaczynała drastycznie się podnosić. Na wykresie 5.1 można zobaczyć oznaczony żółtą linią wynik dla jednego wątku. Jest to ok. 8,5s. Można by spekulować, że procesor radziłby sobie lepiej z większą ilością próbkowania w porównaniu do czasu.
Po dokładniejszej analizie wykresu porównując wyniki 1 mld próbkowania z 3 mld próbek widzimy 3-krotne zwiększenie nakładu na obliczenia, lecz w czasie można zobaczyć, że z 3s nie zwiększył się czas 3-krotnie a lekko poniżej. Oczywiście w takich sytuacjach trzeba także wziąć poprawkę na fakt, że procesor mógł aktualnie być zajęty mniejszą lub większą ilością zadań co na pewno wpłynęło na czas liczenia.
Dodatkowy aspekt, na który można zwrócić uwagę to ilość wątków po której czas zacznie się ponownie zwiększać. Przedstawienie takiego eksperymentu zostało pokazane w tabeli oznaczonej numerem 5.1

| Zestawienie czasu do liczby wątków | | | |
|---|---|---|---|
| Liczba wątków | Czas pomiaru (ms) | | |
| | Liczba Podziału: 100 mln | Liczba Podziału: 1 mld | Liczba Podziału: 3 mld |
| 50 | 38,4 | 353,1 | 1117,9 |
| 100 | 36,1 | 345,3 | 1051,7 |
| 1000 | 52,9 | 353,6 | 1043,7 |
| 10000 | 584,9 | 540,7 | 1331,9 |
| 100000 | 12637,8 | 11920,6 | 11879,5 |

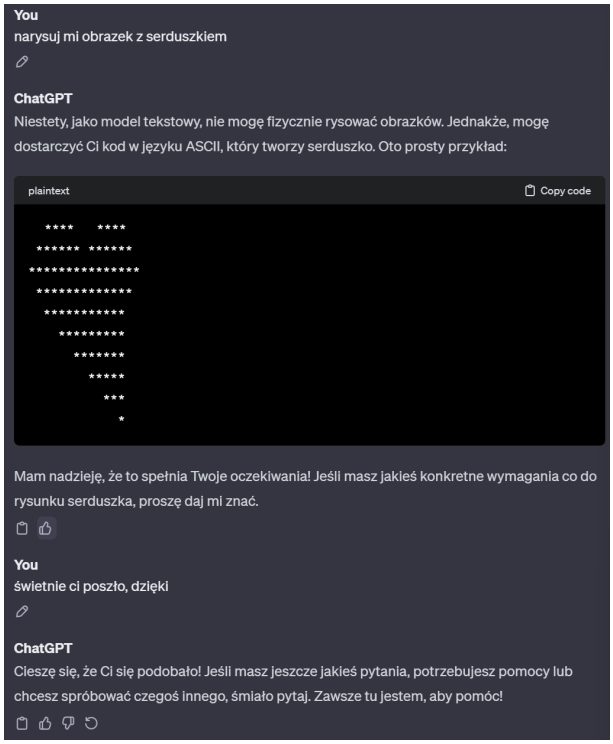
**Tab. 5.1.** Eksperyment ze zwiększoną ilością wątków

Ważną informacją jest że procesor, na którym zostały wykonane testy posiada 12 procesorów logicznych, to znaczy że wszystkie zadania jakie ma do wykonania dzieli między te 12 procesorów logicznych. Oznacza to że przy ilości wątków większej niż 12 czas nie będzie malał, a nawet powinien się zwiększyć. Z ciekawości test przeprowadzony ze znacznie zwiększoną ilością wątków wykazał interesujące wyniki. Między 50-cioma a 100-ma wątkami nie ma większej różnicy, czas dla 100 nawet pokazuje mniejszą ilość czasu, pewnie z powodu akurat zmniejszonej ilości zajęć dla procesora w momencie przeprowadzania testu.
Kolejna ciekawa sprawa to fakt, że do 1000 wątków na to zadanie procesor aż tak się nie różni od poprzedniego testu. Dopiero przy 10 tys. wątków procesor dostaje porządne zajęcie w porównaniu z poprzednim i widzimy nagły wzrost czasu wykonywania.
Wniosek jest dość prosty, zwiększenie ilości wątków ponad siły procesora nie pomoże przy wykonywaniu zadania ale dołoży u pracy, przy sklejaniu wyników z każdego wątku.

# 6. Wnioski

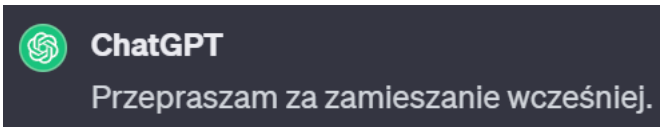
## 6.1. Opis współpracy

Praca ze sztuczną inteligencja taką jak chatGPT w wersji 3.5 jest łatwy, jeśli chodzi o aspekt wymiany informacji. Mamy przez to rozumieć, że "rozmówca"przyjmie wszystkie dane w postaci tekstu bez problemu, szybko je analizuje i wyświetla odpowiedź w postaci tekstu, ekranu z kolorowaniem składni w przypadku generowania kodu czy nawet próby generowania obrazka. Obrazek oczywiście pokaże się nam także w postaci tekstu, mimo to idzie mu nie najgorzej. Przykład możemy zobaczyć na obrazku 6.1. W odpowiedzi na prośbę o obrazek dostaniemy przeprosiny, jendak nie ma co się zamartwiać, ponieważ odpowie z obrazkiem w postaci tekstu.




**Rys. 6.1.** Obrazek narysowany przez chatGPT

Przy dłuższej współpracy niż 10 min zaczniemy zauważać problem sztucznego kolegi polegający na tym, że nie radzi sobie z długimi ciągami tekstu, jeśli dostanie więcej niż ok. 200 wyrazów. W takiej sytuacji często będzie można zobaczyć komunikat przedstawiony na obrazku 6.2.




**Rys. 6.2.** Bot przeprasza

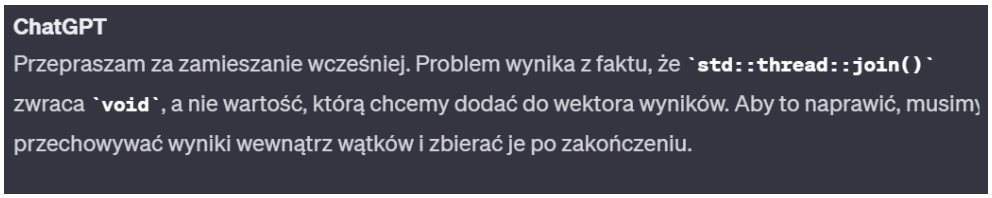
## 6.2. Możliwe pytania

**Czy dobrze się z nim pracuje**

Odpowiedź brzmi: i tak i nie, wszystko zależy czy robimy skomplikowane programy i w jakim języku. Jeśli zależy nam na tworzeniu aplikacji/programów w językach Swift, Go lub TypeScript to bot sam się przyzna że ma mało danych na ten temat. Jesteśmy bezpieczni natomiast w przypadku C, C#, JS, Java, Ruby czy PHP.

**Czy sam napisze program dobrze**

Przez to pytanie rozumie się podanie botowi polecenia i oczekiwanie w 100% działającego programu od samego początku i odpowiedź brzmi: Często tak z drobnym szczegółem. Programy napisane w ten sposób często nie zawierają wszystkich elementów z polecenia i się uruchamiają lub bot popełnia banalne błędy, jeden z przykładów został podany na rysunku 6.3




**Rys. 6.3.** Bot poprawia błąd

**Jak dobrze poprawia kod**

Podczas prowadzenia dłuższej pracy z botem, zdarza mu się być zafiksowanym na jakimś sposobie "naprawy"kodu a często ten element nie jest ani przyczyną problemu ani nawet nie bierze aktywnego udziału w momencie wykonywania programu. Może być to funkcja wykorzystywana przez inną funkcje i podpowie nam że to tam głębiej może być coś nie tak, nawet mimo podania kodu tej funkcji oraz **grzecznie** wytłumaczenia mu że to nie tam jest problem.
Z drugiej strony ma przebłyski i potrafi znaleźć błąd dość szybko i podpowiedzieć rozwiązanie za pierwszym razem. Różnica polega prawdopodobnie na zapisanej w pamięci informacji do której odnosi się bez konkretnego powodu, czasem nawet podając swoje zmiany za niepoprawne mimo, że miały pomóc jeszcze 5 min temu.

**Opisy i pytania dotyczące kodu** W tym miejscu chatGPT świeci jak najjaśniejsza gwiazda. Opisy kodu są trafne, czytelne i zwięzłe. Jeśli chcemy potrafi nawet

edytować napisany przez nas tekst na taki, aby był wpisany do Latexa. Przykładem jest wygenerowanie kodu dla tabeli z danymi czasowymi oznaczonej numerem **??**.

## 6.3. Podsumowanie

W porównaniu z copilotem, chatGPT nie ma dostępu na bieżąco do naszego kodu. Nie uczy się on na tym co piszemy od początku do końca tylko ma dostęp do bazy danych. Używając tych danych analizuje nasze potrzeby i dopasowuje odpowiedzi. Minusem tego jest fakt, że nowe języki wychodzące od zakończenia prowadzenia uczenia się tego bota nie będą uwzględniane. Zbieranie informacji nie zostało mu całkowicie ograniczone, natomiast nauka kodu jest prawdopodobnie zablokowana w tym momencie, dlatego należy czekać na nowe wychodzące wersje i liczyć na to, że pamięć co do prowadzonej konwersacji zostanie poprawiona oraz baza będzie dużo szersza niż dotychczas.

# Bibliografia

[1] *Strona z której można pobrać obrazek PI*. URL: https://freepngimg.com/png/24717-pi-symbol-transparent (term. wiz. 01. 01. 2024).

[2] *Strona internetowa chatGPT*. URL: https://chat.openai.com (term. wiz. 01. 01. 2024).

# Spis rysunków

1.1. Liczba PI . . . . . 3
3.1. Wygenerowany przez bota diagram klas . . . . . 6
4.1. Początek wykonywania obliczeń . . . . . 11
4.2. Zakończenie obliczeń oraz wyświetlenie wyników z czasem obliczeń . . 11
5.1. Wyniki na wykresie . . . . . 14
6.1. Obrazek narysowany przez chatGPT . . . . . 16
6.2. Bot przeprasza . . . . . 17
6.3. Bot poprawia błąd . . . . . 17

# Spis tabel

4.1. Zestawienie czasów pomiaru z liczbami wątków . . . . . 12
4.2. Zestawienie czasów pomiaru z liczbami wątków . . . . . 13
5.1. Eksperyment ze zwiększoną ilością wątków . . . . . 15

# Spis listingów

1. Funkcja calculate_integral . . . . . 8
2. Funkcja thread . . . . . 9
3. Funkcja main . . . . . 9
4. Funkcja main - kontynuacja . . . . . 10