Laboratorium 7: Kwadratury oraz metody Monte Carlo

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

1. **Metoda prostokątów, trapezów oraz Simpsona.**

Zaimplementowałem trzy metody obliczania całki numerycznej.

metoda prostokątów

```
double rectangle_rule(double p, double q, double func(double)){
   double interval = (q-p)/points;
   double result = 0.0;
   for(int i = 0; i < points; i++){
       result += func(p+interval*(i+1))*interval;
   }
   return result;
}
```

metoda trapezów

```
double trapezoidal_rule(double p, double q, double func(double)){
   double interval = (q-p)/points;
   double result = 0.0;
   for(int i = 0; i < points; i++){
       result += (func(p+interval*(i+1))+func(p+interval*i))*interval/2;
   }
   return result;
}
```

metoda Simpsona

```
double simpson_rule(double p, double q, double func(double)){
   double interval = (q-p)/points;
   double result = 0.0;
   double st = 0.0;
   for(int i = 0; i < points; i++){
       double x = p + interval*(i+1);
       st += func(x - interval/2);
       if(i < points-1) result += func(x);
   }
   result = (interval/6) * (func(p) + func(q) + 2*result + 4*st);
   return result;
}
```

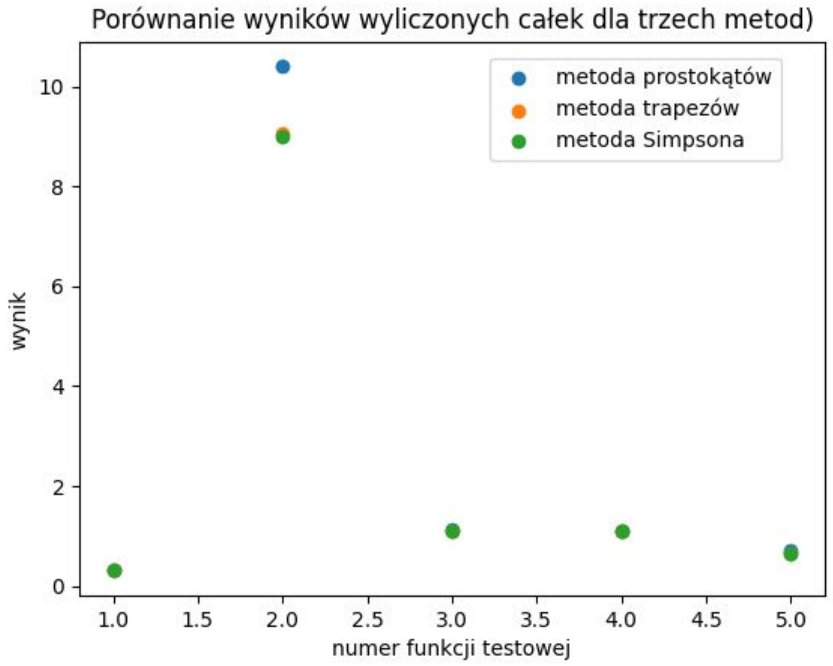
Przetestowałem każdą metodę na pięciu funkcjach. Dokonałem implementacji każdej z testowanych funkcji matematycznych.

```
double f1(double x){ // testowana na przedziale (1, 2)
   return 1/(1+pow(x, 2));
}
```

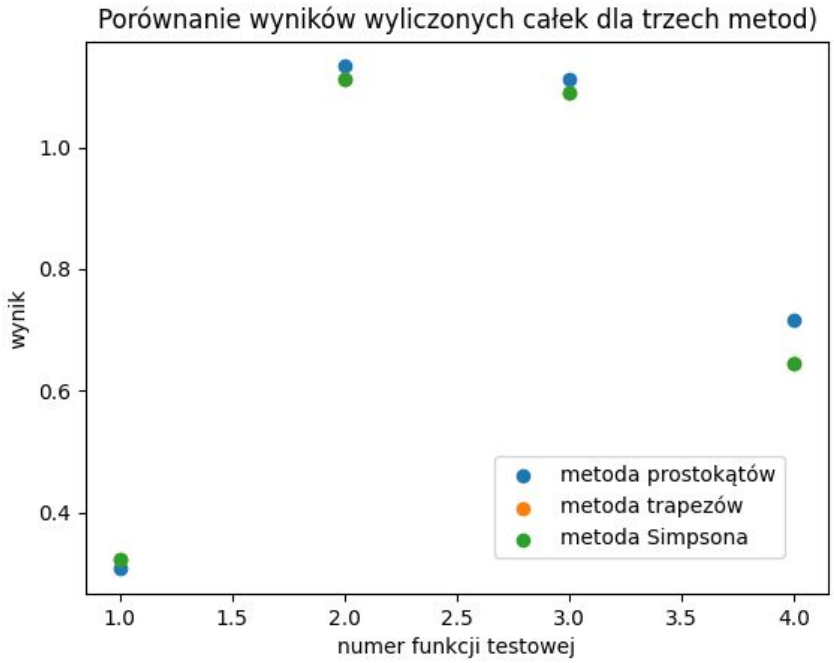
```
double f2(double x){// testowana na przedziale (0, 3)
   return pow(x, 2);
}

double f3(double x){
   return sqrt(pow(x, 3) + 1);// testowana na przedziale (0, 1)
}

double f4(double x){// testowana na przedziale (0, 1)
   return sqrt(pow(x, 4) + 1);
}

double f5(double x){// testowana na przedziale (0, 1)
   return x*sqrt(x + 1);
}
```

Do wyznaczenia rzeczywistego wyniku całki skorzystałem ze strony wolframalpha.com.
W każdej metodzie dzieliłem przedział na 10 równych podprzedziałów. W trakcie testowania metod zauważyłem, że w każdym przypadku metoda Simpsona dawała wynik równy wynikowi wyliczonemu przez wolframalpha.com. Zdecydowałem się nie umieszczać wartości "dokładnego" wyniku na poniższych wykresach - punkty i tak pokryłyby się z punktami metody Simpsona. Numeracja funkcji na osi X wykresów jest zgodna z numeracją funkcji matematycznych w implementacji.

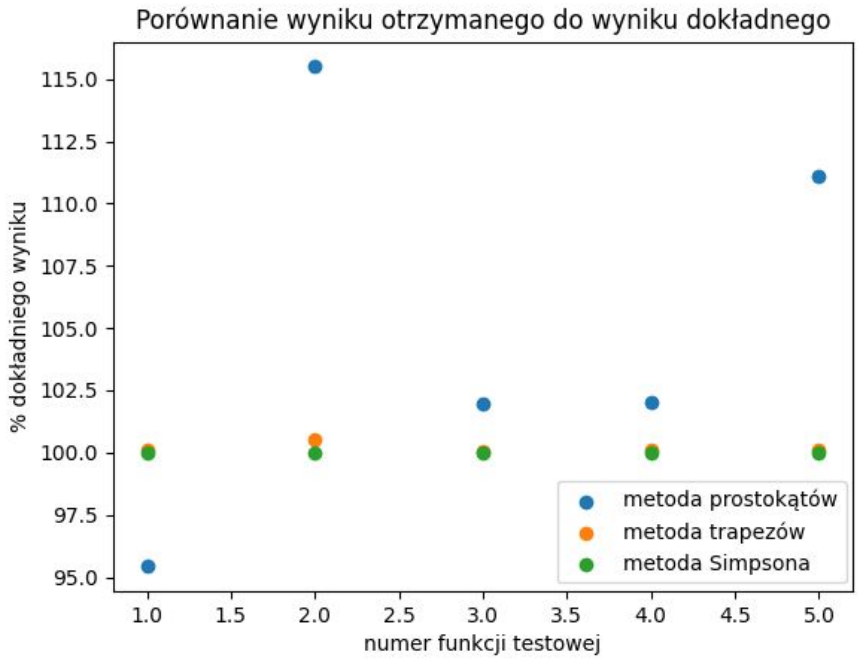
Najpierw utworzyłem wykres, który porównuje dokładne wyniki wszystkich metod.



Wyniki uzyskane podczas całkowania funkcji numer 2 są znacznie większe od pozostałych. To powoduje, że ciężko zauważyć różnice w pozostałych wynikach, gdzie różnice te są bardzo małe. Dlatego zdecydowałem się na utworzenie wykresu dla pozostałych czterech funkcji.



Możemy teraz zauważyć różnice pomiędzy metodą prostokątów, a metodą trapezów i Simpsona. Pomimo niedokładności wyników uzyskanych za pomocą metody trapezów nie możemy ich dostrzec na tym wykresie.
Ostatecznie zdecydowałem się na utworzenie wykresu który przedstawia procentowy stosunek otrzymanych wyników do wyników dokładnych.



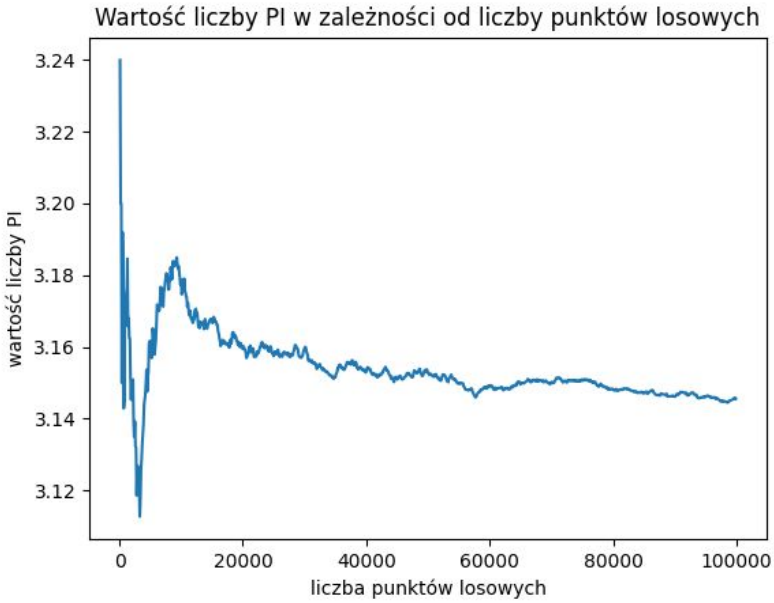
Na ostatnim wykresie można zauważyć niewielkie różnice pomiędzy metodą trapezów, a metodą Simpsona. Można też zauważyć, metoda prostokątów jest najmniej dokładna. W najgorszym badanym przypadku (całka z funkcji x^2 na przedziale (0, 3)) wartość wyznaczona przez metodę prostokątów odbiega od oczekiwanej o ponad 15%.
Metoda trapezów nieznacznie(max 0.5% w testowanych funkcjach) odbiega od oczekiwanych wyników.

2. **Wykorzystanie metody Monte Carlo do obliczenia wartości liczby PI.**

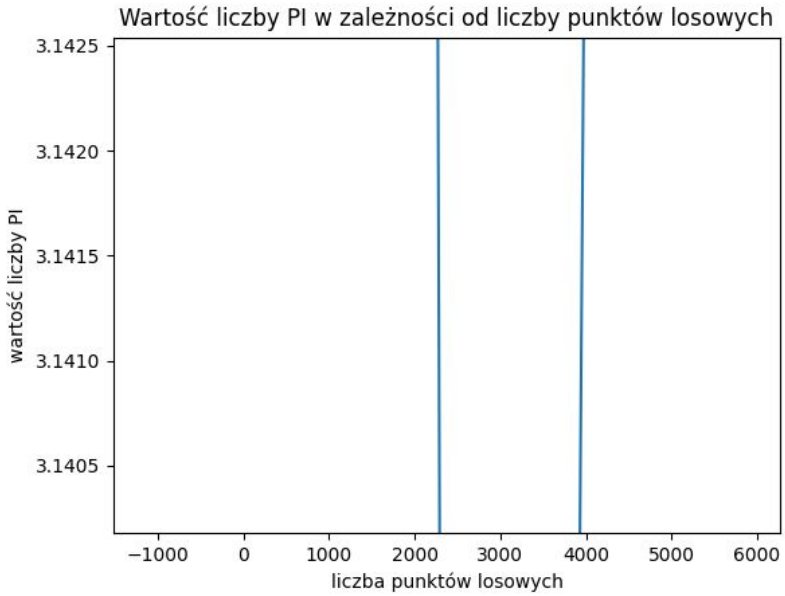
Aby wyznaczyć wartość liczby PI metodą Monte Carlo będę losował punkty w kwadracie o boku 2. W kwadrat ten wpisane jest koło o promieniu 1 i polu powierzchni PI. Stosunek pola koła do pola kwadratu jest równy PI/4, dlatego aby wyznaczyć liczbę PI wystarczy wyliczyć stosunek liczby wylosowanych punktów, które zawiera koło do liczby wszystkich losowanych punktów i stosunek ten pomnożyć przez 4.
Zaimplementowałem metodę, która realizuje ten algorytm:

```
double get_pi(int points_number){
    std::default_random_engine gen;
    std::uniform_real_distribution<double> random{-1.0, 1.0};
    int hits = 0;
    for(int i = 0; i < points_number; i++){
        double x = random(gen);
        double y = random(gen);
        if(x*x + y*y <= 1) { hits++; }
    }
    std::cout << "PI: " << (float(hits))/(points_number/4) << std::endl;
}
```

Wyznaczyłem wartość liczby pi w zależności od liczby punktów losowych. Pomiary dokonałem iterując od 100 do 100000 punktów z krokiem iteracji równym 100 punktów. Wyniki przedstawiłem na wykresie:



Wartość najbardziej przybliżona do rzeczywistej została otrzymana dwa razy. Przybliżając wykres można zauważyć, że otrzymano ją dla około 2200 i 3900 punktów losowych.



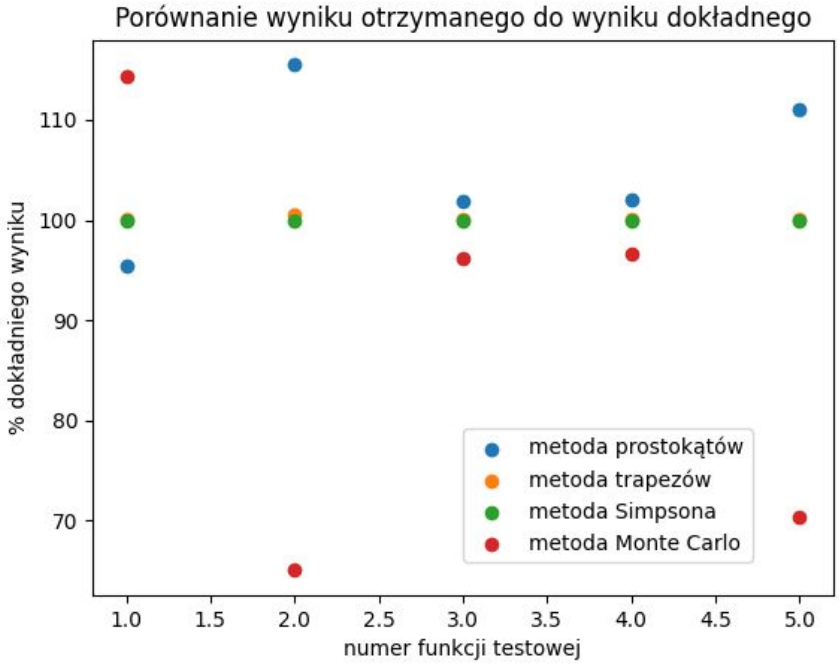
Z pierwszego wykresu wynika też, że dokładność wyznaczonej liczby PI nie zawsze wzrasta wraz z liczbą punktów losowych. Prawdopodobnie wynika to z ograniczonej liczby punktów w cyklu generatora.

3. Obliczanie całki numerycznej metodą Monte Carlo

Zaimplementowałem metodę Monte Carlo do wyznaczania całki numerycznej. Metoda jest bardzo prosta - oblicza średnią wartość funkcji w określonej liczbie punktów. Następnie, aby wyliczyć wartość całki należy przemnożyć wartość średnią funkcji przez długość przedziału.

```
double monte_carlo_rule(double p, double q, double func(double)){
   double result = 0.0;
   std::default_random_engine gen;
   std::uniform_real_distribution<double> random{p, q};
   for(int i = 0; i < points; i++){
       double x = random(gen);
       result += func(x);
   }
   result /= points;
   result *= (q-p);
   return result;
}
```

Dokonałem pomiarów na tych samych danych testowych, co w przypadku pierwszych trzech metod. Aby wyniki były porównywalne użyłem takiej samej liczby punktów pomiaru(10). Otrzymane wyniki podzieliłem przez dokładne i pomnożyłem przez 100%.



Metoda Monte Carlo nie jest dokładna. Otrzymane wyniki mają najgorszą dokładność z wszystkich czterech metod. Jest za to szybka w implementacji i łatwa w zrozumieniu.

Zwiększenie liczby punktów pomiaru w badanej metodzie do 10000 minimalizuje wartość błędu do 0.07% we wszystkich badanych przypadkach. Taka wartość błędu często jest akceptowalna, dlatego metoda Monte Carlo może być dobrym rozwiązaniem np. gdy potrzebujemy jednorazowo obliczyć całkę i nie zależy nam na dokładności, ani na lekko dłuższym(zazwyczaj nieodczuwalnie) czasie wykonania.