Laboratorium 9: Równania różniczkowe zwyczajne

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

1. **Metody rozwiązywania równań różniczkowych**

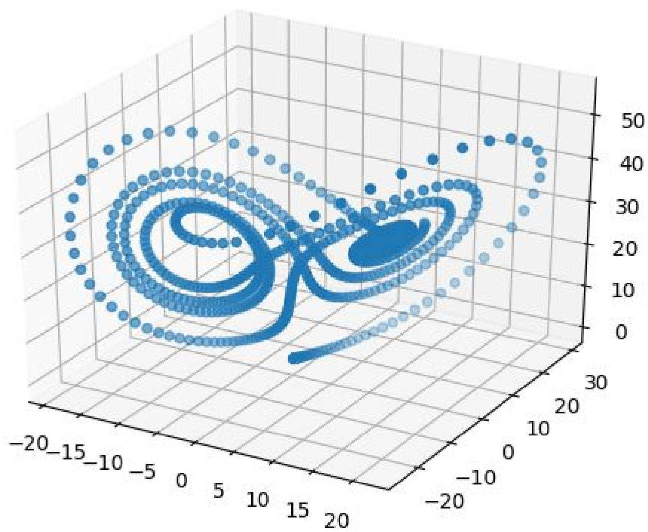
Dokonałem implementacji czterech metod rozwiązywania równań różniczkowych (dla układu Lorenza). Poniżej zamieszczam funkcje pomocnicze oraz zmienne globalne które użyłem w implementacji.

```
#define SIGMA 10
#define RHO 28
#define BETA 2.6666
#define DELTA 0.01
#define ITERATIONS 1000

double dxdt(double x, double y){
   return SIGMA * (y - x);
}

double dydt(double x, double y, double z){
   return x * (RHO - z) - y;
}

double dzdt(double x, double y, double z){
   return x * y - BETA * z;
}
```

metoda Eulera aka metoda Rungego-Kutty 1 rzędu

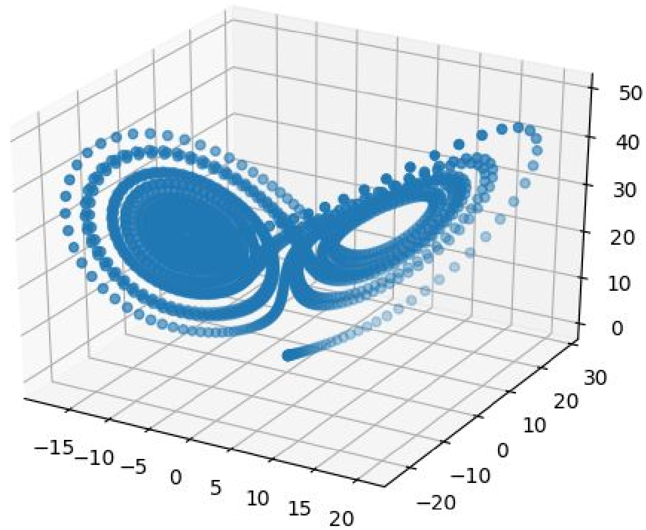
```
void eulerMethod(double x, double y, double z){
   for(int i = 0; i < ITERATIONS; i++){
       double xt = x + DELTA * dxdt(x, y);
       double yt = y + DELTA * dydt(x, y, z);
       double zt = z + DELTA * dzdt(x, y, z);
//         std::cout << x << std::endl;
//         std::cout << y << std::endl;
       std::cout << z << std::endl;

       x = xt;
       y = yt;
       z = zt;
   }
}
```

Otrzymany wynik(dla danych początkowych x = 0.1, y = 0, z = 0)



metoda Eulera(implicit)

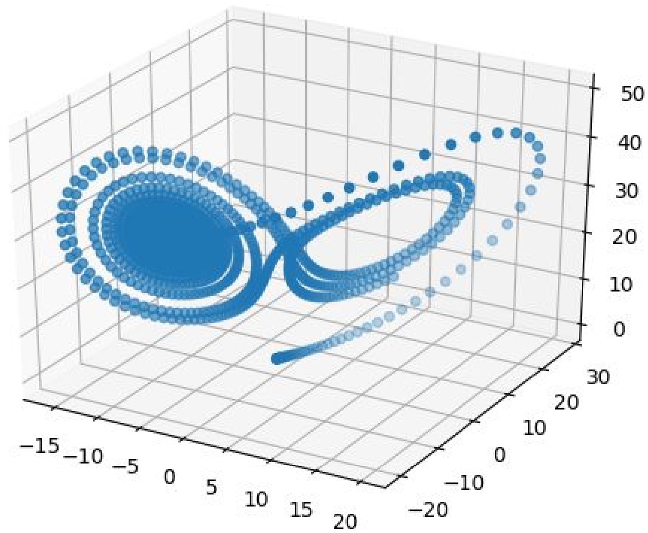
```
void eulerMethodBackward(double x, double y, double z){
   for(int i = 0; i < ITERATIONS; i++){
       double xt = x + DELTA * dxdt(x, y);
       double yt = y + DELTA * dydt(xt, y, z);
       double zt = z + DELTA * dzdt(xt, yt, z);
       x = xt;
       y = yt;
       z = zt;
       std::cout << x << std::endl;
       std::cout << y << std::endl;
       std::cout << z << std::endl;
   }
}
```



metoda Rungego-Kutty 4-tego rzędu

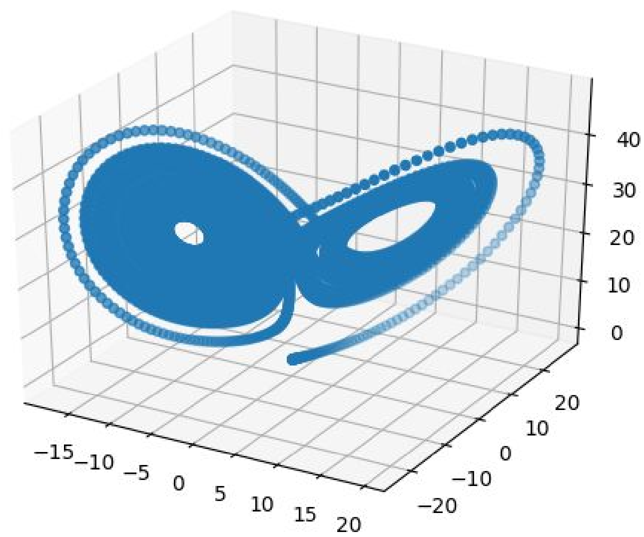
```
double f1(point3 p, point3 pn){
   return SIGMA * (p.y  + pn.y * DELTA / 2 - (p.x + pn.x * DELTA / 2));
}

double f2(point3 p, point3 pn){
   return -1 * (p.x + pn.x * DELTA / 2) * (p.z + pn.z * DELTA / 2) + RHO *
(p.x + pn.x * DELTA / 2) - (p.y + pn.y * DELTA / 2);
}

double f3(point3 p, point3 pn){
   return (p.x + pn.x * DELTA / 2) * (p.y + pn.y * DELTA / 2) - BETA * (p.z
+ pn.z * DELTA / 2);
}

point3 getK1(){
   point3 result;
   point3 zero;
   zero.x = zero.y = zero.z = 0;
   result.x = f1(old, zero);
   result.y = f2(old, zero);
   result.z = f3(old, zero);
   return result;
}

point3 getKn(point3 p){
```

```
    point3 result;
    result.x = f1(old, p);
    result.y = f2(old, p);
    result.z = f3(old, p);
    return result;
}

point3 getNextPoint(point3 startPoint){
    old = startPoint;

    point3 k1 = getK1();
    point3 k2 = getKn(k1);
    point3 k3 = getKn(k2);
    point3 k4 = getKn(k3);

    point3 nextPoint;
    nextPoint.x = old.x + (k1.x + 2 * k2.x + 2 * k3.x + k4.x) * DELTA / 6;
    nextPoint.y = old.y + (k1.y + 2 * k2.y + 2 * k3.y + k4.y) * DELTA / 6;
    nextPoint.z = old.z + (k1.z + 2 * k2.z + 2 * k3.z + k4.z) * DELTA / 6;

    return nextPoint;
}

void rk4(point3 startPoint){
    for(int i = 0; i < ITERATIONS; i++){
        point3 nextPoint = getNextPoint(startPoint);
        std::cout << nextPoint.x << std::endl;
        std::cout << nextPoint.y << std::endl;
        std::cout << nextPoint.z << std::endl;
        startPoint = nextPoint;
    }
}
```



metoda Rungego-Kutty 2-ego rzędu

```
point3 getNextPointRK2(point3 startPoint){
   old = startPoint;

   point3 k1 = getK1();
   point3 k2 = getKn(k1);

   point3 nextPoint;
   nextPoint.x = old.x + (k1.x + 2 * k2.x) * DELTA / 6;
   nextPoint.y = old.y + (k1.y + 2 * k2.y) * DELTA / 6;
   nextPoint.z = old.z + (k1.z + 2 * k2.z) * DELTA / 6;
   return nextPoint;
}


void rk2(point3 startPoint){
   for(int i = 0; i < ITERATIONS; i++){
       point3 nextPoint = getNextPointRK2(startPoint);
       std::cout << nextPoint.x << std::endl;
       std::cout << nextPoint.y << std::endl;
       std::cout << nextPoint.z << std::endl;
       startPoint = nextPoint;
   }
}
```



2. Porównanie teoretyczne metod

Metoda Eulera jest najbardziej prymitywną metodą z wszystkich przedstawionych w tym sprawozdaniu. Polega ona na odczytywaniu wartości szukanej funkcji w równoodległych punktach podanej dziedziny.
Metodę Eulera można w prosty sposób zmodyfikować - podczas wyliczania kolejnych współrzędnych punktu możemy korzystać z wartości które zostały już wyznaczone. Chociaż rząd złożoności obliczeniowej pozostaje taki sam to w ten sposób uzyskujemy szybszą zbieżność metody do wyniku, ponieważ korzystamy z bardziej aktualnych danych.
Metody Rungego-Kutty są jednymi z najczęściej stosowanych metod w rozwiązywaniu równań różniczkowych. Każda z metod korzysta z funkcji pomocniczej zależnej od czasu, poprzedniego punktu oraz wartości skoku pomiędzy kolejnymi punktami. Podczas wyznaczania kolejnego punktu metody korzystają z rozwinięcia w szereg Taylora wokół ostatnio wyznaczonego punktu. Rząd metody Rungego-Kutty jest równoznaczny z ilością wyrazów z rozwinięcia Taylora które bierzemy pod uwagę w obliczeniach. Na tej podstawie łatwo stwierdzić, że wraz ze wzrostem rzędu metody wzrasta też jej dokładność. Obliczanie kolejnych pochodnych jest jednak kłopotliwe.
Z racji, że dla pierwszego rzędu bierzemy tylko pierwszą pochodna funkcji pomnożoną przez krok iteracyjny - metoda ta jest równoznaczna z metodą Eulera.

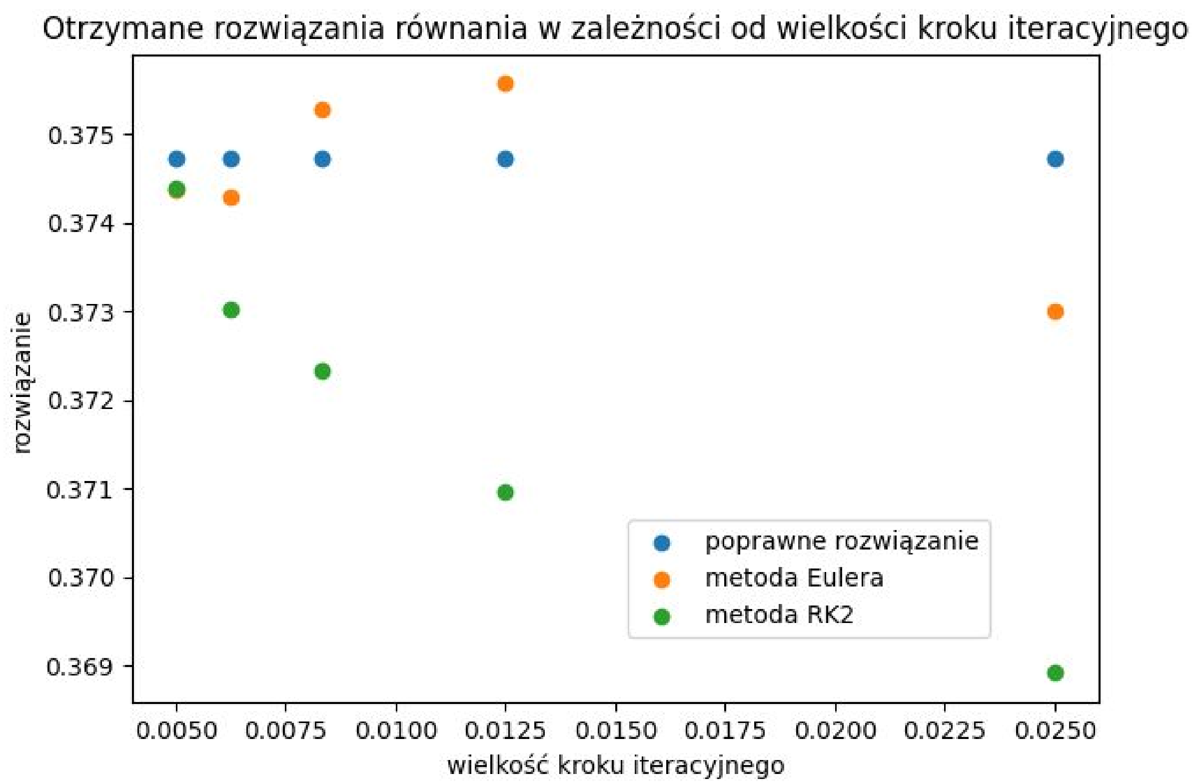
3. **Porównanie wyznaczonego wyniku z wynikiem poprawnym**

Zaimplementowałem funkcje które rozwiązują podane w zadaniu równanie metodą Eulera i Rungego-Kutty 2-ego rzędu. Założyłem, że k = m = 1.
Zaimplementowałem również funkcję która wyznacza dokładny wynik równania.

```
double dydx(double x, double y){
   return y * sin(x) + sin(x) * cos(x);
}

double func(double x){
   return exp(-cos(x)) - cos(x) + 1;
}

void euler(double x0, double xk, double y, double h){
   double tmp;
   while(x0 < xk){
       tmp = y;
       y = y + h * dydx(x0, y);
       x0 = x0 + h;
   }
   std::cout << y << std::endl;
}


double newValue(point3 p, double h){
   return dydx(old.x, old.y) + (dydx(p.x, p.y) * h / 2);
}

point3 iterate(point3 startPoint, double h){
   old = startPoint;
   point3 zero;
   zero.x = zero.y = zero.z = 0;

   point3 nextPoint;
   point3 k1;
   point3 k2;
   k1.x = k2.x = old.x + h;
   k1.y = newValue(zero, h);
   k2.y = newValue(k1, h);

   nextPoint.x = k1.x;
   nextPoint.y = old.y + (k1.y + 2 * k2.y) * DELTA / 6;
   return nextPoint;
}
```

```
point3 solveEquation(point3 startPoint, double h, double xk){
   while(startPoint.x < xk){
       point3 next = iterate(startPoint, h);
       startPoint = next;
   }
   return startPoint;
}
```

Następnie wyznaczyłem rozwiązanie dla pięciu różnych wielkości kroku iteracyjnego. Wyniki przedstawia poniższy wykres.



Zmniejszenie kroku iteracyjnego powoduje zwiększenie dokładności otrzymanego rozwiązania. Można było się tego spodziewać - im mniejszy krok tym większa ilość wykonywanych iteracji. Metoda Rungego-Kutty jest w większości badanych przypadków mniej dokładna. Dla h = 0.005 metoda Eulera zwraca minimalnie gorszy wynik.