Julia Plewa
MOwNiT
Laboratorium 8

1. **Proszę zapoznać się z materiałami załączonymi do tego laboratorium (zwłaszcza z opisami metod!).**

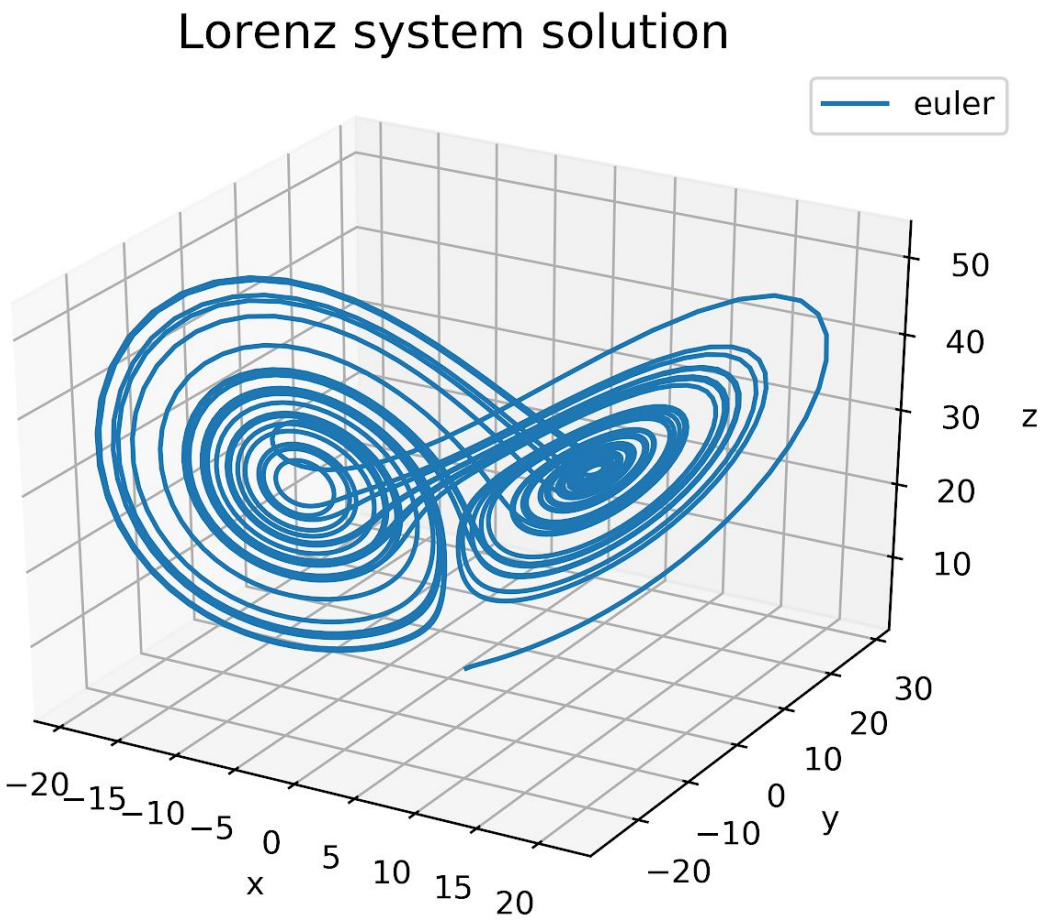
2. **Proszę wykonać implementacje metod rozwiązywania równań różniczkowych. Za ich pomocą rozwiązać opisany powyżej układ równań. Wynik wyliczeń kolejnych kroków proszę przedstawić na wykresie (powinno się otrzymać powyższy atraktor). Środowisko i język tworzenia wykresu jest dowolny.**

   a. **metoda Eulera,**

   kod:

```
class Euler : IDifferentialEquation
{
   public:
   Euler(LorenzSystem lorenzSystem, std::vector<double> initialState) :
      IDifferentialEquation(lorenzSystem, initialState) {}

   std::vector<std::vector<double>> solve(double t0, double h, int n)
   {
       std::vector<std::vector<double>> results;
       double x = initialState[0];
       double y = initialState[1];
       double z = initialState[2];
       double t = t0;

       for (int j = 0; j < n; j++)
       {
           std::vector<double> derivatives = lorenzSystem.GetDerivatives(x, y, z);
           x = x + h * derivatives[0];
           y = y + h * derivatives[1];
           z = z + h * derivatives[2];
           t += h;
           results.push_back({x, y, z, t});
       }
       return results;
   }
};
```

rezultat:



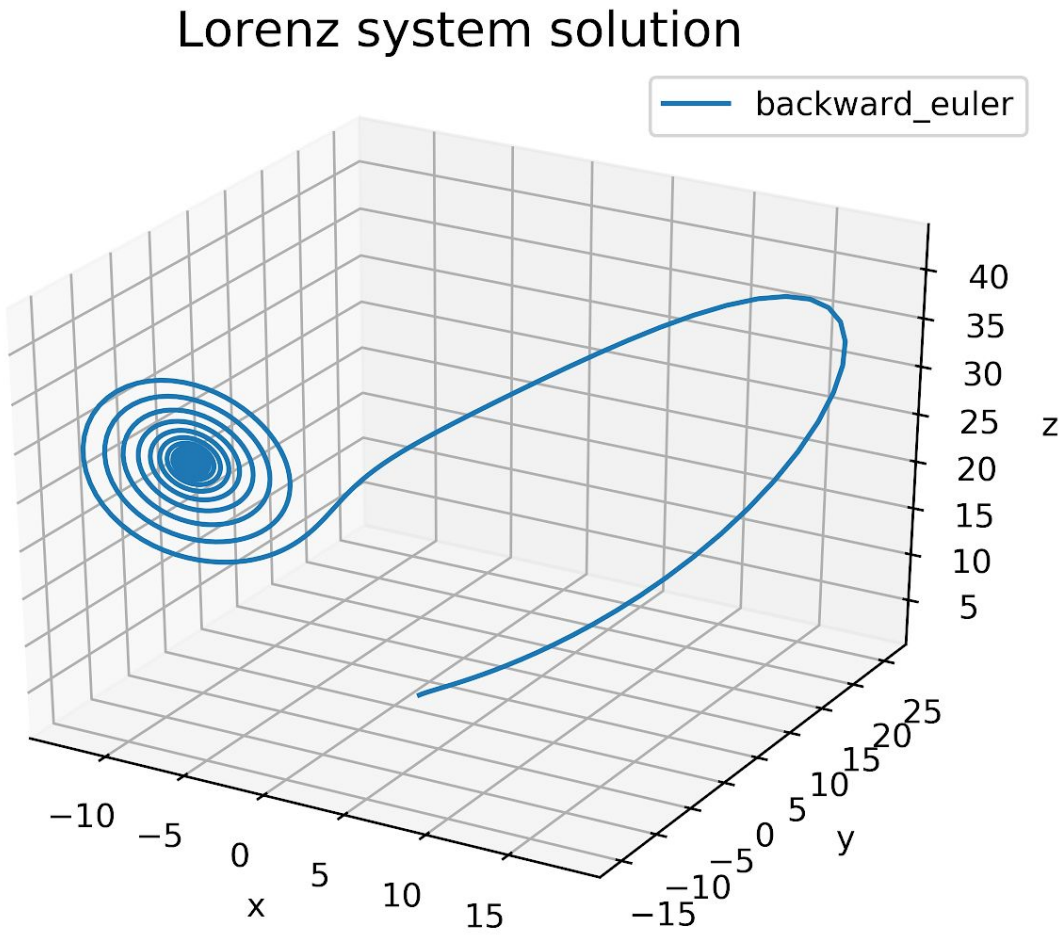
b. modyfikacja metody Eulera (ang. Backward Euler method),

kod:

```
class BackwardEuler : IDifferentialEquation
{
   public:
   BackwardEuler(LorenzSystem lorenzSystem, std::vector<double> initialState) :
     IDifferentialEquation(lorenzSystem, initialState) {}

   std::vector<std::vector<double>> solve(double t0, double h, int n)
   {
       std::vector<std::vector<double>> results;
       double x = initialState[0];
       double y = initialState[1];
       double z = initialState[2];
       double t = t0;
```

```
        for (int j = 0; j < n; j++)
        {
            t += h;
            std::vector<double> derivatives = approximateDerivatives(x, y, z, h);
            x = derivatives[0];
            y = derivatives[1];
            z = derivatives[2];
            results.push_back({x, y, z, t});
        }
        return results;
    }

    private:
    std::vector<double> approximateDerivatives(double x, double y,
       double z, double h)
    {
        std::vector<double> derivatives = lorenzSystem.GetDerivatives(x, y, z);
        double x0 = x + h * derivatives[0];
        double y0 = y + h * derivatives[1];
        double z0 = z + h * derivatives[2];

        derivatives = lorenzSystem.GetDerivatives(x0, y0, z0);
        double x1 = x + h * derivatives[0];
        double y1 = y + h * derivatives[1];
        double z1 = z + h * derivatives[2];

       for (int i = 0; i < FIXED_POINT - 2; i++)
            derivatives = lorenzSystem.GetDerivatives(x1, y1, z1);

            x0 = x1;
            y0 = y1;
            z0 = z1;

            x1 = x + h * derivatives[0];
            y1 = y + h * derivatives[1];
            z1 = z + h * derivatives[2];
        }
        return {x1, y1, z1};
    }
};
```

rezultat:



c. metoda Rungego-Kutty 1 rzędu (It's a trap!),

Metoda Rungego-Kutty rzędu 1 jest równoważna z metodą Eulera.

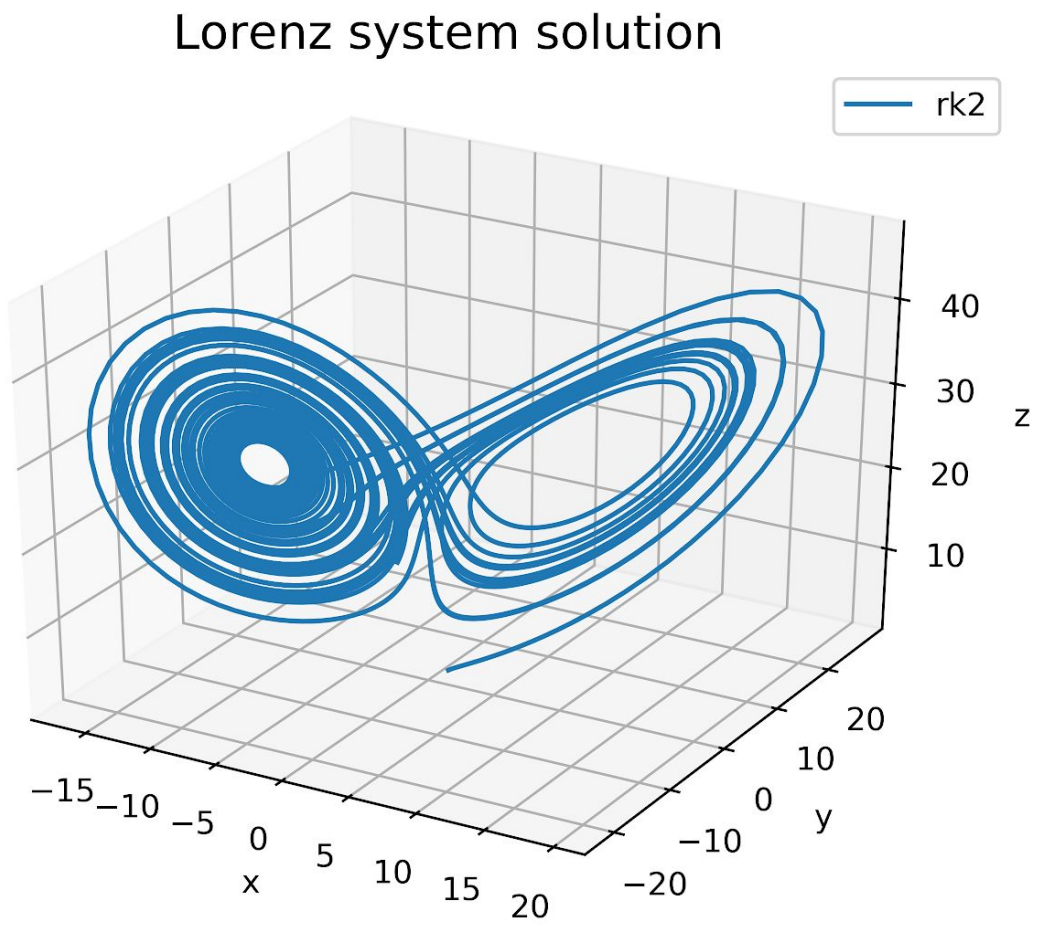
d. metoda Rungego-Kutty 2 rzędu (ang. midpoint method),

kod:

```
class RungeKutta2 : IDifferentialEquation
{
   public:
   RungeKutta2(LorenzSystem lorenzSystem, std::vector<double> initialState) :
      IDifferentialEquation(lorenzSystem, initialState) {}

   std::vector<std::vector<double>> solve(double t0, double h, int n)
   {
      std::vector<std::vector<double>> results;
      double x = initialState[0];
      double y = initialState[1];
      double z = initialState[2];
      double t = t0;
```

```
        for (int j = 0; j < n; j++)
        {
            std::vector<double> derivatives = lorenzSystem.GetDerivatives(x, y, z);
            derivatives = lorenzSystem.GetDerivatives(x + h * derivatives[0] / 2,
                y + h * derivatives[1] / 2,
                z + h * derivatives[2] / 2);
            x = x + h * derivatives[0];
            y = y + h * derivatives[1];
            z = z + h * derivatives[2];
            t += h;
            results.push_back({x, y, z, t});
        }
        return results;
    }
};
```

rezultat:



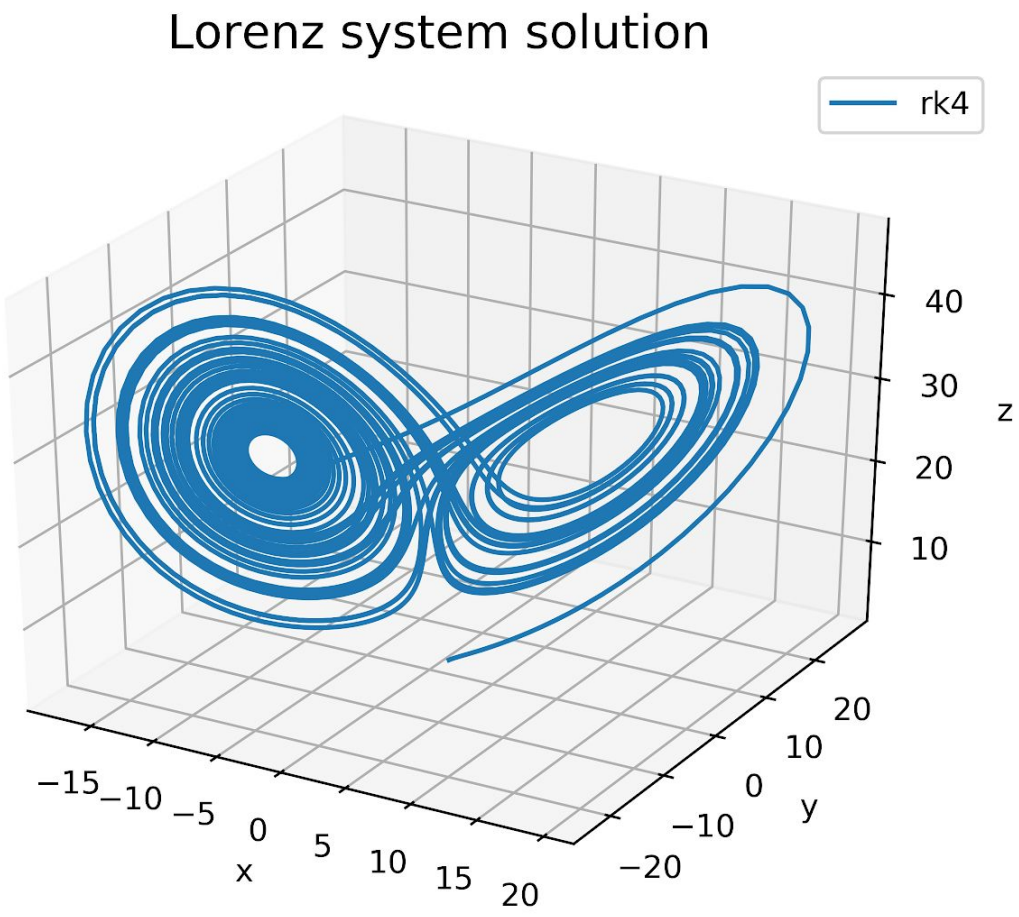
e. metoda Rungego-Kutty 4 rzędu.

kod:

```
class RungeKutta4 : IDifferentialEquation
{
 public:
   RungeKutta4(LorenzSystem lorenzSystem, std::vector<double> initialState) :
      IDifferentialEquation(lorenzSystem, initialState) {}

   std::vector<std::vector<double>> solve(double t0, double h, int n)
   {
       std::vector<std::vector<double>> results;
       double x = initialState[0];
       double y = initialState[1];
       double z = initialState[2];
       double t = t0;

       for (int j = 0; j < n; j++)
       {
          double k_x, k_y, k_z, sum_x, sum_y, sum_z;

          // k1
          std::vector<double> derivatives = lorenzSystem.GetDerivatives(x, y, z);
          k_x = h * derivatives[0];
          k_y = h * derivatives[1];
          k_z = h * derivatives[2];
          sum_x = k_x;
          sum_y = k_y;
          sum_z = k_z;

          // k2
          derivatives = lorenzSystem.GetDerivatives(x + k_x / 2, y + k_y / 2,
             z + k_z / 2);
          k_x = h * derivatives[0];
          k_y = h * derivatives[1];
          k_z = h * derivatives[2];
          sum_x += 2 * k_x;
          sum_y += 2 * k_y;
          sum_z += 2 * k_z;
```

```
            // k3
            derivatives = lorenzSystem.GetDerivatives(x + k_x / 2, y + k_y / 2,
               z + k_z / 2);
            k_x = h * derivatives[0];
            k_y = h * derivatives[1];
            k_z = h * derivatives[2];
            sum_x += 2 * k_x;
            sum_y += 2 * k_y;
            sum_z += 2 * k_z;

            // k4
            derivatives = lorenzSystem.GetDerivatives(x + k_x, y + k_y, z + k_z);
            k_x = h * derivatives[0];
            k_z = h * derivatives[2];
            k_y = h * derivatives[1];
            sum_x += k_x;
            sum_y += k_y;
            sum_z += k_z;

            x = x + sum_x / 6;
            y = y + sum_y / 6;
            z = z + sum_z / 6;
            t += h;
            results.push_back({x, y, z, t});
        }
        return results;
    }
};
```

rezultat:



3. **Proszę dokonać porównania teoretycznego wszystkich powyższych metod.**

Metody Rungego-Kutty to metody numerycznego rozwiązywania równań różniczkowych pierwszego rzędu w postaci: $y' = f(t,\ y)$.

Metoda Rungego-Kutty rzędu r jest zdefiniowana wzorem

$$y_{i+1} = y_i + h \cdot \sum_{i=1}^{r} c_i k_i(t,\ y,\ h), \quad i = 0, 1, ..., N-1, \quad \text{gdzie}$$

$$k_i(t,\ y,\ h) = f\left(t + h \cdot \sum_{j=1}^{r} b_{ij},\ y + h \cdot \sum_{j=1}^{r} b_{ij} k_j \right).$$

Powyżej przedstawione zostały implementacje metod Rungego-Kutty rzędu 1 (*metoda Eulera*), rzędu 2 (*midpoint method*) oraz rzędu 4. Ponadto zaimplementowano wariant metody Eulera *backward Euler*.

Wyższe rzędy metod Rungego-Kutty w dużym uproszczeniu prowadzą do dokładniejszych rozwiązań, choć jest to tak naprawdę zależne od właściwości danego układu. RK4 jest powszechnie stosowane ze względu na prostotę implementacji i jakość działania.

Backward Euler jest przykładem metody typu *implicit* czyli takiej, w której do obliczenia stanu układu w przyszłości wykorzystywany jest sam poszukiwany stan z przyszłości. Pozostałe zaimplementowane metody są typu *explicit*. Tego typu metody nie są stosowne do rozwiązywania układów równań sztywnych (czyli takich, których rozwiązania pewnymi metodami numerycznymi mimo małego kroku całkowania są niestabilne). Dla układów równań niesztywnych (takich jak układ Lorenza) korzysta się raczej właśnie z metod *explicit*. Nie wymagają one korzystania dodatkowo z algorytmów do znalezienia pierwiastków funkcji takich jak *metoda Newtona-Raphsona* czy tzw. *fixed-point iteration*.

Przewaga metod *implicit* nad *explicit* jest dostrzegalna w przypadku wspomnianych równań sztywnych - aby osiągnąć dobrej jakości rozwiązanie nie ma konieczności drastycznego zmniejszania kroku. Pomimo konieczności wykonywania dodatkowych obliczeń jest to opłacalne.

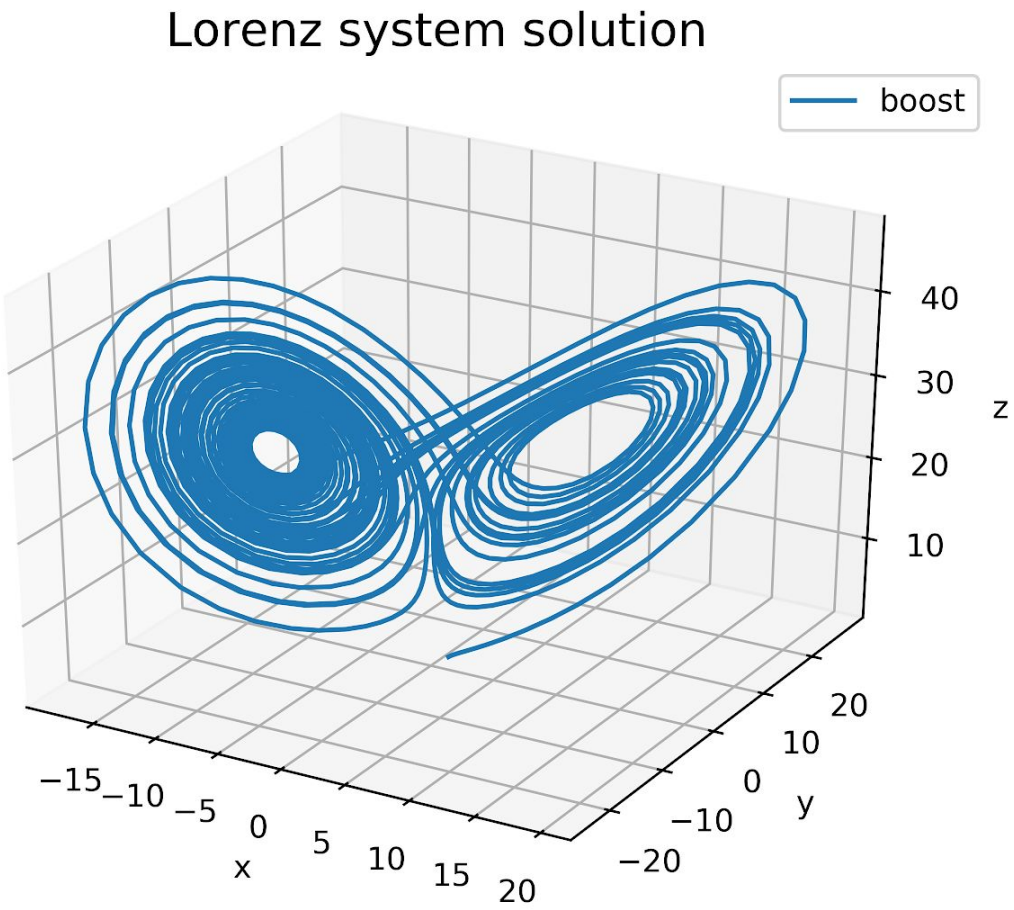
Na powyższych wykresach rozwiązań łatwo zauważyć, że rozwiązanie układu Lorenza metodą *implicit* wyraźnie się różni - rozwiązanie traci swój chaotyczny charakter wynikający z zależności kolejnych wyników od stanu początkowego i zbiega do stanu równowagi.

4. **Proszę rozwiązać układ Lorenza korzystając z funkcjonalności biblioteki boost.**

kod:

```
class BoostSolver : IDifferentialEquation
{
   public:
   BoostSolver(LorenzSystem lorenzSystem, std::vector<double> initialState) :
      IDifferentialEquation(lorenzSystem, initialState) {}

      std::vector<std::vector<double>> solve(double t0, double h, int n)
      {
         std::function<void(stateType, stateType&, double)> der =
            [&](stateType x, stateType &dxdt, double t)
         {
            std::vector<double> derivatives = this ->
               lorenzSystem.getDerivatives(x[0], x[1], x[2]);
            dxdt[0] = derivatives[0];
            dxdt[1] = derivatives[1];
            dxdt[2] = derivatives[2];
         };
         std::function<void(stateType, double)> save = [](stateType x, double t)
         {
            appendSolution(x, t);
         };
         stateType state = {initialState[0], initialState[1], initialState[2]};
         integrate(der, state, t0, (double) n * h, h, save);
         return this -> solution;
      }

      void appendSolution(stateType x, double t)
      {
         solution.push_back({x[0], x[1], x[2], t});
      }

      private:
      std::vector<std::vector<double>> solution;
};
```

rezultat:



5. **Proszę ze zrozumieniem zapoznać się i opisać algorytm Verleta w wariancie algorytmu skokowego (ang. leap-frog algorithm).**

   Algorytm Verleta jest metodą numeryczną wykorzystywaną do całkowania równań ruchu (czyli do obliczania położeń i prędkości układu oddziałujących ciał w funkcji czasu). Pozwala rozwiązać równania różniczkowe zwyczajne rzędu drugiego i mówiąc w pewnym uproszczeniu zachowuje energię układu (w odróżnieniu od metod takich jak metoda Rungego-Kutty), równocześnie nie będąc bardziej kosztownym od klasycznego algorytmu Eulera. Znajduje zastosowanie w symulacjach fizycznych ruchu cząsteczek oraz w grafice komputerowej. Jego najpopularniejsze warianty to algorytm podstawowy, algorytm prędkościowy oraz algorytm skokowy (*leap-frog algorithm*). Choć są one równoważne matematycznie, to w różny sposób radzą sobie ze propagacją błędów zaokrągleń.

Algorytm skokowy pozwala na numeryczne rozwiązanie równania w postaci:

$$\widehat{v} = \frac{dv}{dt} = F(x),\ \widehat{x} = \frac{dx}{dt} = v\ .$$

W i-tym kroku wartości położenia i prędkości przyjmują następujące wartości:

$$\begin{aligned} x_i &= x_{i-1} + v_{i-\frac{1}{2}}\Delta t\,, \\ a_i &= F(x_i), \\ v_{i+\frac{1}{2}} &= v_{i-\frac{1}{2}} + a_i\Delta t\,. \end{aligned}$$

Dla zachowania stabilności algorytmu Δt powinno być wartością stałą.