MOwNiT2 Lab8

Zadanie 2 Proszę wykonać implementacje następujących metod rozwiązywania równań różniczkowych:

- metoda Eulera,
- modyfikacja metody Eulera (ang. Backward Euler method),
- metoda Rungego-Kutty 1 rzędu (It's a trap!),
- metoda Rungego-Kutty 2 rzędu (ang. midpoint method),
- metoda Rungego-Kutty 4 rzędu,

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.

Klasa, w której zawarto wszystkie implementacje, przygotowana do rozwiązywania układu Lorenza:

```
namespace LorenzSystem
{

      class Solver
  {
    public:
      Solver(double rho, double sigma, double beta, double dt, int max_steps) :
        rho(rho), sigma(sigma), beta(beta), dt(dt), max_steps(max_steps)  {}

  // methods..

    private:
      std::vector<double> getValues(double x, double y, double z){
        std::vector<double> result;
        result.push_back(sigma * y - sigma * x);
        result.push_back(-x * z + rho * x - y);
        result.push_back(x * y - beta * z);
        return result;
      }

    private:
      double rho;
      double sigma;
      double beta;
      double dt;
      int max_steps;
  };

}
```

1. Metoda Eulera (Rungego-Kutty 1 rzędu):

```cpp
void eulerMethod(double x, double y, double z){

  int step = 1;
  std::vector<double> result;
  std::ofstream outfile("explicit_euler.csv");

  while(step <= max_steps){

    result = getValues(x, y, z);
    x = x + result[0] * dt;
    y = y + result[1] * dt;
    z = z + result[2] * dt;

    outfile << x << "," << y << "," << z << std::endl;

    step++;
  }

}
```

2. Backward Euler:

```
void backwardEulerMethod(double x, double y, double z){

  double err = 1e-6;
  int step = 1;
  std::vector<double> result;
  std::ofstream outfile("implicit_euler.csv");

  while(step <= max_steps){

    result = getValues(x, y, z);
    double x0 = x + dt * result[0];
    double y0 = y + dt * result[1];
    double z0 = z + dt * result[2];

    result = getValues(x0, y0, z0);
    double x1 = x + dt * result[0];
    double y1 = y + dt * result[1];
    double z1 = z + dt * result[2];

    while(abs(x0-x1) >= err || abs(y0-y1) >= err || abs(z0-z1) >= err){

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

      result = getValues(x0, y0, z0);
      double x1 = x + dt * result[0];
      double y1 = y + dt * result[1];
      double z1 = z + dt * result[2];
    }

    x = x1;
    y = y1;
    z = z1;

    outfile << x << "," << y << "," << z << std::endl;

    step++;
  }

}
```

3. Metoda Rungego-Kutty 2 rzędu:

```
void midpointMethod(double x, double y, double z){

  int step = 1;
  std::vector<double> result;
  std::ofstream outfile("midpoint.csv");

  while(step <= max_steps){

    result = getValues(x, y, z);
    double tmpx = x + dt/2 * result[0];
    double tmpy = y + dt/2 * result[1];
    double tmpz = z + dt/2 * result[2];

    result = getValues(tmpx, tmpy, tmpz);
    x = x + result[0] * dt;
    y = y + result[1] * dt;
    z = z + result[2] * dt;

    outfile << x << "," << y << "," << z << std::endl;

    step++;
  }

}
```

4. Metoda Rungego-Kutty 4 rzędu:

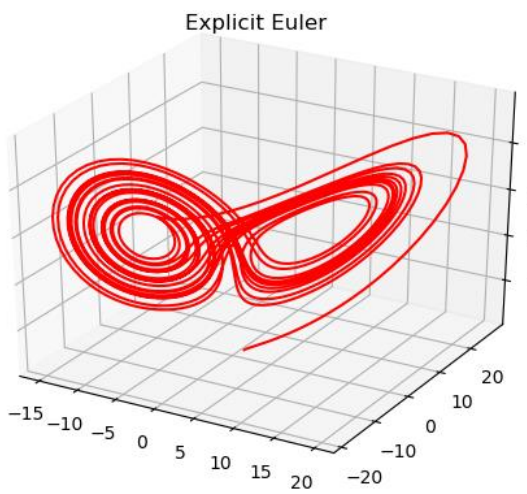
```
void rungeKuttaMethod(double x, double y, double z){

  std::vector<double> k1, k2, k3, k4;
  int step = 1;
  std::ofstream outfile("runge_kutta.csv");

  while(step <= max_steps){

    k1 = getValues(x, y, z);
    for(int i=0; i<3; i++) k1[i] *= dt;

    k2 = getValues(x+k1[0]/2, y+k1[1]/2, z+k1[2]/2);
    for(int i=0; i<3; i++) k2[i] *= dt;

    k3 = getValues(x+k2[0]/2, y+k2[1]/2, z+k2[2]/2);
    for(int i=0; i<3; i++) k3[i] *= dt;

    k4 = getValues(x+k3[0], y+k3[1], z+k3[2]);
    for(int i=0; i<3; i++) k4[i] *= dt;

    x = x + (k1[0] + 2*k2[0] + 2*k3[0] + k4[0])/6;
    y = y + (k1[1] + 2*k2[1] + 2*k3[1] + k4[1])/6;
    z = z + (k1[2] + 2*k2[2] + 2*k3[2] + k4[2])/6;


    outfile << x << "," << y << "," << z << std::endl;

    step++;
  }

}
```

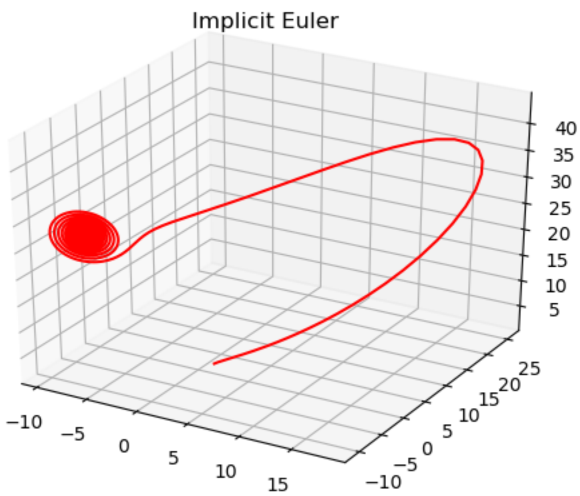
$$\frac{dx}{dt} = \sigma y - \sigma x$$
$$\frac{dy}{dt} = -xz + \rho x - y$$
$$\frac{dz}{dt} = xy - \beta z$$

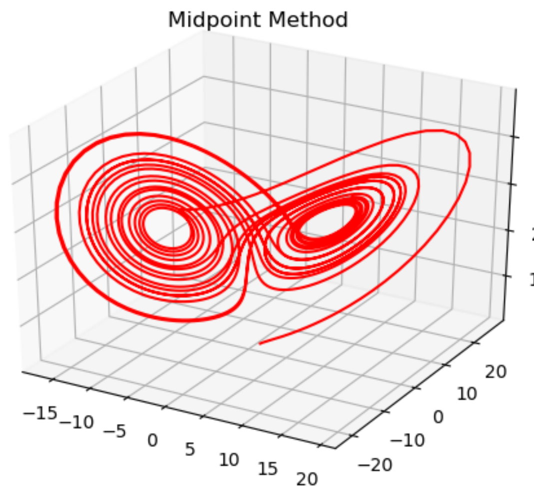
Atraktor Lorenza z powyższego układu możemy otrzymać dla wartości:

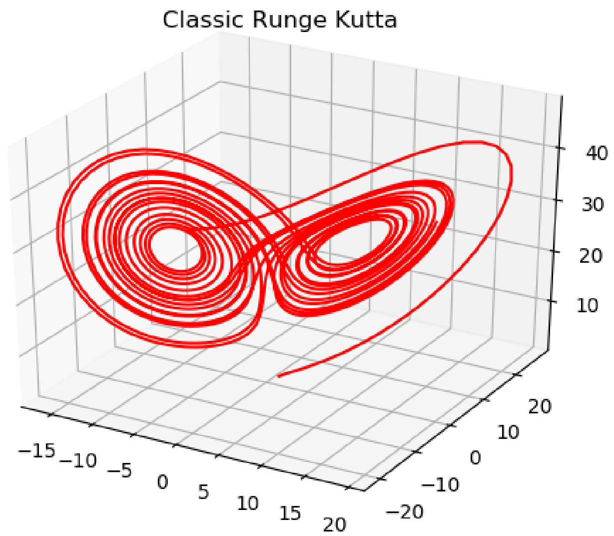
$$\sigma = 10, \rho = 28, \beta = 8/3$$

Przykładowe atraktory dla 2500 kroków, dt=0.01
(Wykresy wygenerowane w pythonie):









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

Definiujemy funkcję f jako prawą stronę równania różniczkowego:
$\frac{dy}{dt} = f(t, y)$,
h jako wielkość kroku oraz $y(t_0) = y_0$ jako wartość początkową.

Nasz układ jest autonomiczny więc nie jest wymagane przekazywane zmiennej czasu przy wykonywaniu obliczeń.

1. Metoda Eulera (Rungego-Kutty 1 rzędu):

- Najprostsza ze wszystkich metod
- Mało efektywna
- Niestabilna numerycznie (jedyna z tych czterech)
- Wzór:

$$y_{n+1} = y_n + f(t_n, y_n) * h$$

2. Backward Euler:

- Odpowiednia dla równań sztywnych, w przeciwieństwie do metody Eulera
- Mała modyfikacja wzoru względem metody Eulera
- Wzór:

$$y_{n+1} = y_n + f(t_{n+1}, y_{n+1}) * h$$

- Wymaga rozwiązania równania algebraicznego ze względu na dwa wystąpienia y_{n+1}

3. Metoda Rungego-Kutty 2 rzędu (Midpoint Method):

- Znana jako „zmodyfikowana metoda Eulera”
- Korzysta ze środkowej wartości pomiędzy kolejnymi krokami
- Wzór:

$$y_{n+1} = y_n + f(t_n + \frac{h}{2}, y_n + \frac{h}{2} f(t_n, y_n)) * h$$

4. Metoda Rungego-Kutty 4 rzędu:

- Metoda klasyczna
- Wylicza wartość za pomocą średniej ważonej czterech składowych: wartości z początku przedziału, 2 wartości ze środka dla różnych argumentów oraz wartości z końca przedziału.
- Wzór:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

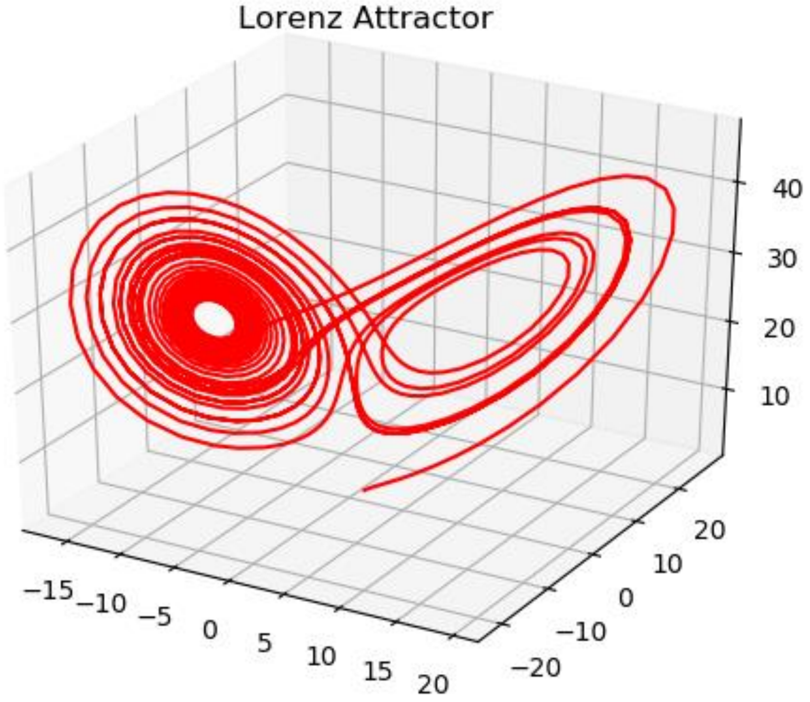
$$k_1 = f(t_n, y_n) * h$$
$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) * h$$
$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) * h$$
$$k_4 = f(t_n + h, y_n + k_3) * h$$

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

```
#include <fstream>
#include <vector>
#include <boost/numeric/odeint.hpp>


const double sigma = 10.0;
const double rho = 28.0;
const double beta = 8.0 / 3.0;

typedef std::vector<double> state;
std::ofstream outfile("boost.csv");

void lorenzSystem(const state &x, state &dxdt, double ){

 dxdt[0] = sigma * x[1] - sigma * x[0];
 dxdt[1] = -x[0] * x[2] + rho * x[0] - x[1];
 dxdt[2] = x[0] * x[1] - beta * x[2];
}

void saveCoord(const state &x, const double ){

 outfile << x[0] << "," << x[1] << "," << x[2] << std::endl;
}

int main(void){

 state x0 = {1.0, 1.0, 1.0};
 double t0 = 0.0, t = 25.0, dt = 0.1;
 boost::numeric::odeint::integrate(lorenzSystem, x0, t0, t, dt, 
                                    saveCoord);
}
```

Tutaj funkcja integrate() korzysta z metody Dormand-Prince do rozwiązywania równań różniczkowych zwyczajnych.

Atraktor za pomocą biblioteki boost:



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

- Jedna z metod drugiego rzędu
- Służy do rozwiązywania równań typu:

$$\frac{dx^2}{d^2t} = F(x)$$

- Z powyższego widać, że odpowiada to liczeniu przyspieszenia ($a = \frac{dx^2}{d^2t}$), prędkości ($v = \frac{dx}{dt}$) i położenia (x) w danym czasie (t).
- Wzór:

$$a_{i+1} = F(x_{i+1})$$
$$v_{i+1/2} = v_{i-1/2} + a_i * \Delta t$$
$$x_{i+1} = x_i + v_{i+1/2} * \Delta t$$

- Nazwa pochodzi od faktu, że wartości prędkości i położenia są wyliczane naprzemiennie
- Wersja zsynchronizowana:

$$a_{i+1} = F(x_{i+1})$$
$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1}) * \Delta t$$
$$x_{i+1} = x_i + v_i * \Delta t + \frac{1}{2} a_i * \Delta t^2$$

- Algorytm jest odwracalny w czasie
- Zachowuje energię całkowitą bliską jej początkowej wartości