AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody Obliczeniowe w Nauce i Technice:

Równania różniczkowe zwyczajne

Laboratorium 9

Przemysław Lechowicz

1. Układ Lorenza

Układ Lorenza jest to przedstawiony układ trzech nieliniowych równań różniczkowych zwyczajnych modelujący w możliwie najprostszy sposób zjawisko konwekcji termicznej w atmosferze.

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x),$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y,$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z.$$

Przyjmuję, że:

$$\rho = 28, \beta = \frac{8}{3}, \sigma = 10$$

2. Zadanie 2

Proszę wykonać implementację następujących metod rozwiązywania równań różniczkowych:

a. metoda Eulera

Wykonałem implementację klasy realizującą metodę Eulera dla układu Lorenza:

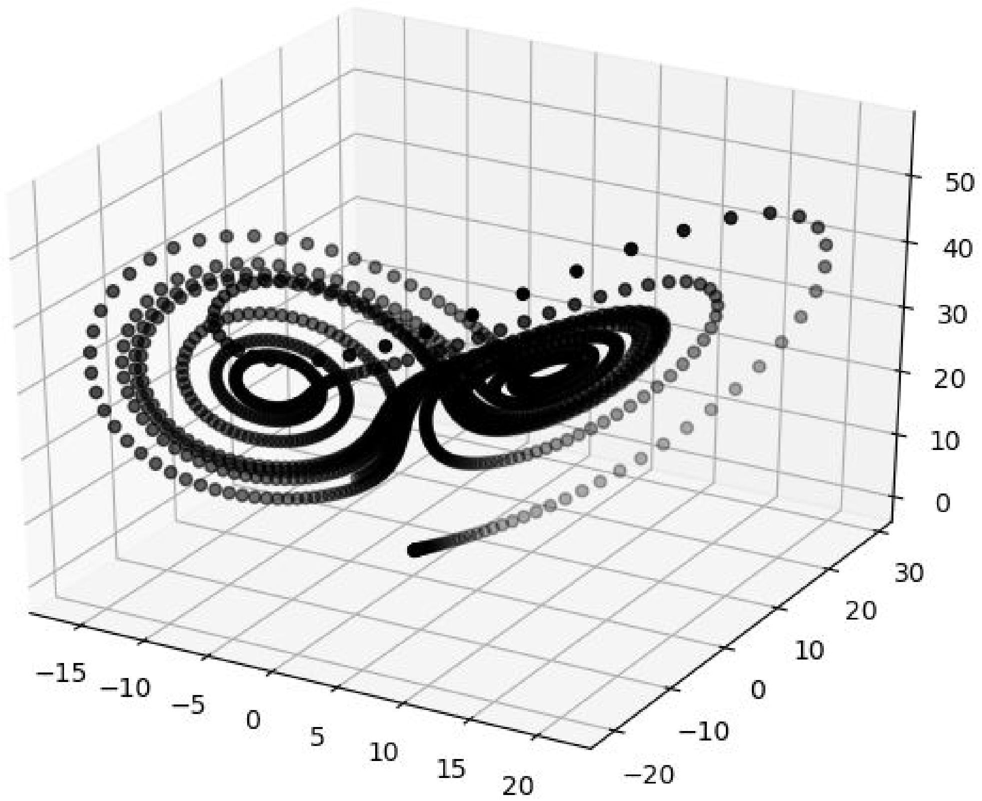
```
class Lorenz {
private:
	double sigma;
	double rho;
	double beta;
	double x;
	double y;
	double z;
	int IterationCount;
public:
	Lorenz(int iterationCount, double sigma = 10, double rho = 28, double beta = 8 /
3, double x = 0.1, double y = 0, double z = 0) {
		this->IterationCount = iterationCount;
		this->beta = beta;
		this->sigma = sigma;
		this->rho = rho;
		this->x = x;
		this->y = y;
		this->z = z;
	}

	void euler() {
		double step = 0.01;
		std::ofstream myfile;
```

```
                myfile.open("euler.txt");
                for (int i = 0; i < this->IterationCount; i++) {
                        double xt = x + step * this->sigma * (y - x);
                        double yt = y + step * (x * (this->rho - z) - y);
                        double zt = z + step * (x * y - this->beta * z);
                        x = xt;
                        y = yt;
                        z = zt;
                        myfile << x <<  std::endl;
                        myfile << y <<  std::endl;
                        myfile << z <<  std::endl;
                }
                myfile.close();
        }
};
```

Sama wizualizacja atraktora została wykonana przy użyciu języka python z biblioteką matplotlib:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

f = open("euler.txt", "r")
x = []
y = []
z = []
num_lines = sum(1 for line in open('euler.txt'))

for i in range(int(num_lines / 3)):
   x.append(float(f.readline()))
   y.append(float(f.readline()))
   z.append(float(f.readline()))

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(x, y, z, c='black', marker='o')
plt.show()
```

Wynikiem działania tego programu jest następujący atraktor:

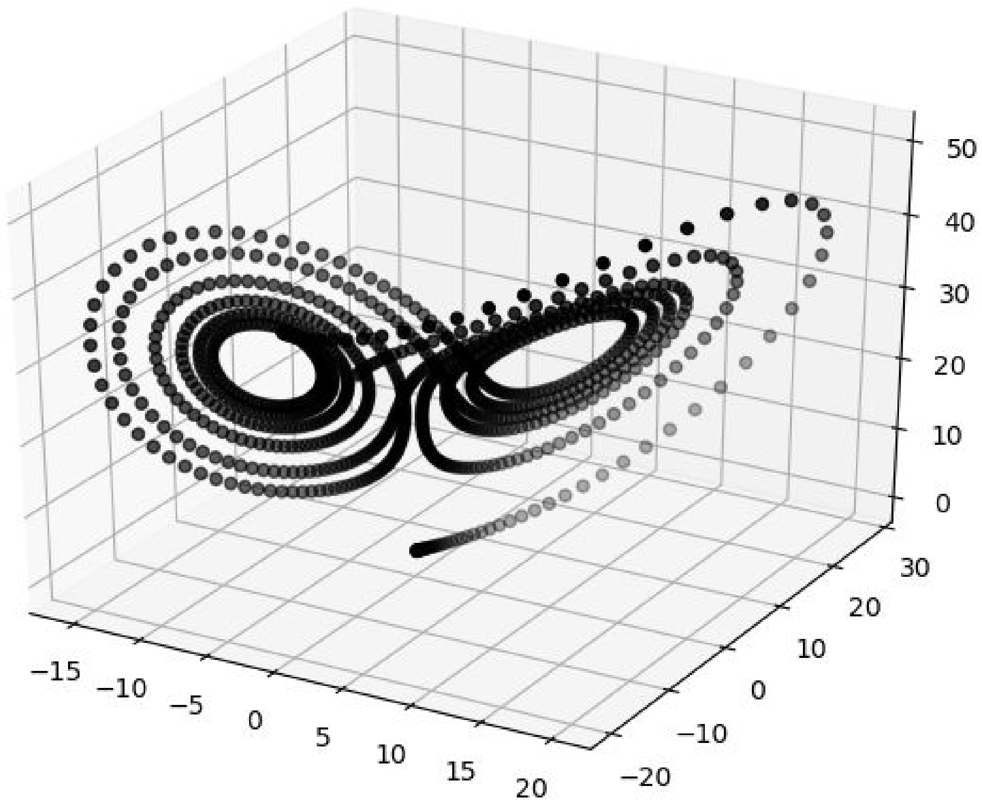


b. modyfikacja metody Eulera (ang. Backward Euler method)

Zaimplementowana została metoda Backward Euler method:

```
void backwardEuler() {
        double step = 0.01;
        std::ofstream myfile;
        myfile.open("backwardEuler.txt");
        for (int i = 0; i < this->IterationCount; i++) {
                double xt = x + step * this->sigma * (y - x);
                double yt = y + step * (xt * (this->rho - z) - y);
                double zt = z + step * (xt * yt - this->beta * z);
                x = xt;
                y = yt;
                z = zt;
                myfile << x << std::endl;
                myfile << y << std::endl;
                myfile << z << std::endl;
        }
        myfile.close();
}
```

Otrzymany atraktor:



c. metoda Rungego-Kutty 1 rzędu

Metoda Eulera jest szczególnym przypadkiem metod Runge-Kutty (metoda Eulera pojawiła się historycznie najpierw, w związku z tym zachowano tradycyjną nazwę) - Metoda Rungego-Kutty 1 rzędu jest metodą Eulera.

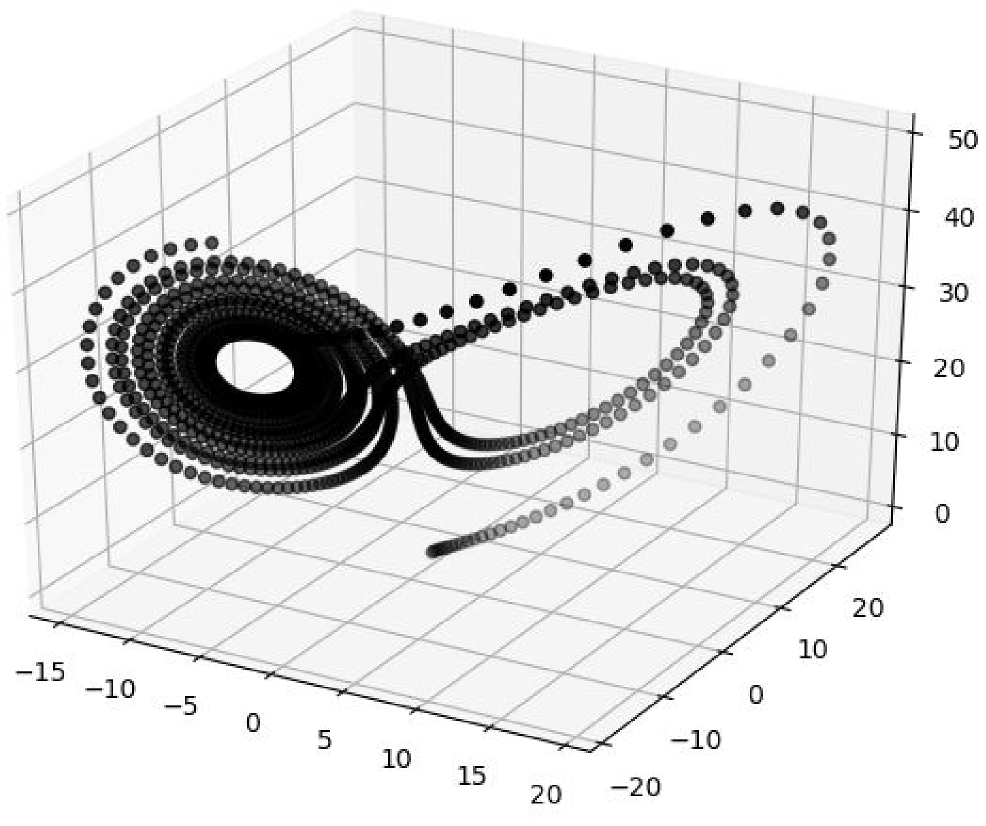
d. metoda Rungego-Kutty 2 rzędu

Zaimplementowana została metoda Rungego-Kutty 2 rzędu:

```
void rungeKutta() {
        double step = 0.01;
        std::ofstream myfile;
        system("rm rungekutta.txt");
        myfile.open("rungekutta.txt");
        x = 1;
        y = 1;
        z = 1;
        for (int i = 1; i < this->IterationCount + 1; i++) {
                double k1 = sigma * (y - x);
                double l1 = x * rho - x * z - y;
                double m1 = x * y - beta * z;
```

```
            double k2 = sigma * ((y + (0.5 * l1 * step)) - (x + 0.5 * k1 * step));
            double l2 = (x + 0.5 * k1 * step) * rho - (x + 0.5 * k1 * step) * (z +
(0.5 * m1 * step)) - (y + (0.5 * l1 * step));
            double m2 = (x + 0.5 * k1 * step) * (y + (0.5 * l1 * step)) - beta * (z +
(0.5 * m1 * step));

            x = x + step * k2;
            y = y + step * l2;
            z = z + step * m2;

            myfile << x << std::endl;
            myfile << y << std::endl;
            myfile << z << std::endl;
        }
        myfile.close();
}
```

Otrzymany atraktor:

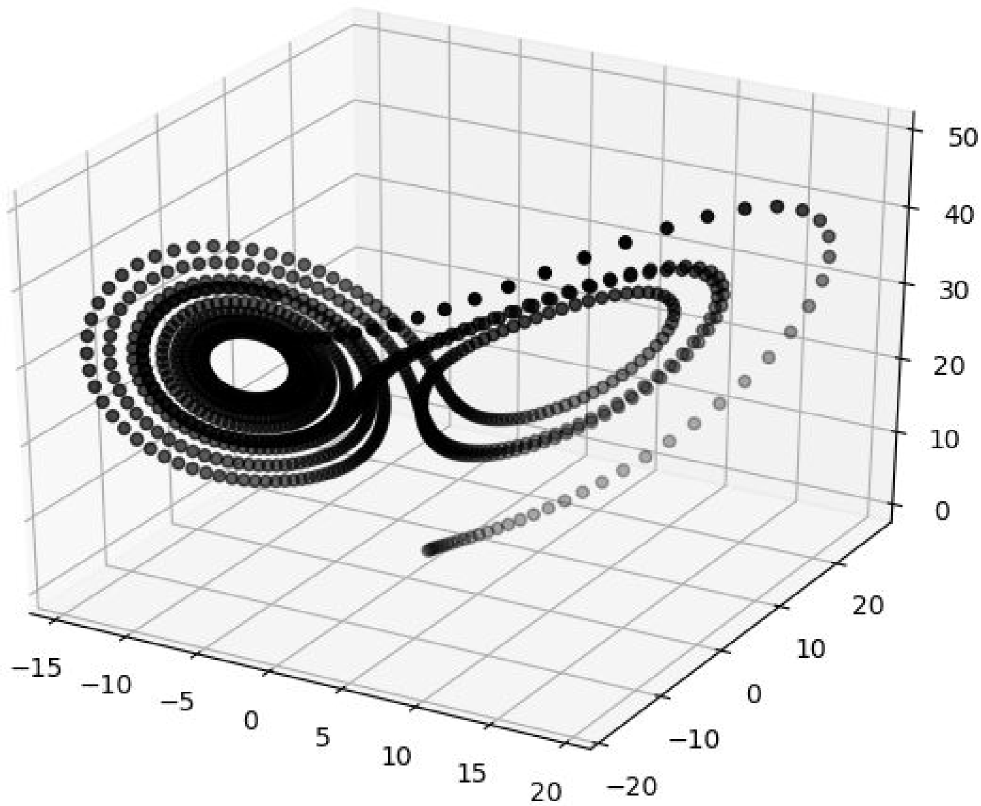


e. metoda Rungego-Kutty 4 rzędu

Zaimplementowana została metoda realizująca metodę Rungego-Kutty 4-rzędu:

```
void rk4() {
	double step = 0.01;
	std::ofstream myfile;
	myfile.open("rk4.txt");
	x = 1;
	y = 1;
	z = 1;
	for (int i = 1; i < this->IterationCount + 1; i++) {
		double k1 = sigma * (y - x);
		double l1 = x * rho - x * z - y;
		double m1 = x * y - beta * z;

		double k2 = sigma * ((y + (0.5 * l1 * step)) - (x + 0.5 * k1 * step));
		double l2 = (x + 0.5 * k1 * step) * rho - (x + 0.5 * k1 * step) * (z +
(0.5 * m1 * step)) - (y + (0.5 * l1 * step));
		double m2 = (x + 0.5 * k1 * step) * (y + (0.5 * l1 * step)) - beta * (z +
(0.5 * m1 * step));

		double k3 = sigma * ((y + (0.5 * l1 * step)) - (x + 0.5 * k1 * step));
		double l3 = (x + 0.5 * k1 * step) * rho - (x + 0.5 * k1 * step) * (z +
(0.5 * m1 * step)) - (y + (0.5 * l1 * step));
		double m3 = (x + 0.5 * k1 * step) * (y + (0.5 * l1 * step)) - beta * (z +
(0.5 * m1 * step));

		double k4 = sigma * ((y + l3 * step) - (x + k3 * step));
		double l4 = (x + k3 * step) * rho - (x + k3 * step) * (z + m3 * step) - (y
+ l3 * step);
		double m4 = (x + k3 * step) * (y + l3 * step) - beta * (z + m3 * step);

		x = x + step * (k1 + 2 * k2 + 2 * k3 + k4) / 6;
		y = y + step * (l1 + 2 * l2 + 2 * l3 + l4) / 6;
		z = z + step * (m1 + 2 * m2 + 2 * m3 + m4) / 6;

		myfile << x << std::endl;
		myfile << y << std::endl;
		myfile << z << std::endl;
	}
	myfile.close();
}
```

Otrzymany atraktor:



3. Zadanie 3

Proszę dokonać porównania teoretycznego wszystkich powyższych metod.

Różnicą pomiędzy metodą Eulera, a odwrotną metodą Eulera jest kolejność wyliczania kolejnych danych. W metodzie Eulera kierujemy się wzorem:

$$y_{n+1} = y_n + hf(t_n, y_n).$$

Podczas gdy w odwrotnej metodzie Eulera stosujemy wzór:

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}).$$

Jako metodę Rungego-Kutty 2. rzędu wykorzystałem metodę punktu pośredniego, która korzysta ze wzoru:

$$y_{n+1} = y_n + hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right).$$
,

gdzie:

$$k_1 = hf\left(x_n, y_n\right),$$

Natomiast metoda Rungego-Kutty 4. rzędu korzysta z następujących wzorów:

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

gdzie:

$$k_1 = hf(x_n, y_n) ,$$
$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right) ,$$
$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_2\right) ,$$
$$k_4 = hf(x_n + h, y_n + k_3) .$$

W ten sposób otrzymujemy, podobnie jak w innych iteracyjnych metodach rozwiązywania równań różniczkowych kolejne punkty, które przybliżają rozwiązanie.

Metody typu Runge-Kutty (w tym również 1. rzędu - metoda Eulera) są łatwe w implementacji, zmiana kroku całkowania może być dokonywana w dowolnym etapie obliczeń i nie wymaga dużego nakładu pracy. Wymagają one jednak wielokrotnego (dla metody rzędu p p-krotnego) obliczania wartości funkcji f w każdym kroku całkowania, mogą więc być metodami kosztownymi (zwykle najbardziej czasochłonnymi, a więc i kosztownym zadaniem jest obliczanie wartości funkcji f). Wszystkie mają złożoność czasową O(n), gdzie n jest liczbą iteracji.

Rząd metody	Wartości współczynników	Nazwa metody	Błąd lokalny	Błąd całkowity
1	$k_1 = hf(x_n, y_n) ,$	Eulera	$O(h^2)$	$O(h)$
2	$k_1 = hf(x_n, y_n) ,$ $k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right) ,$	punktu pośredniego	$O(h^3)$	$O(h^2)$
4	$k_1 = hf(x_n, y_n) ,$ $k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right) ,$ $k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_2\right) ,$ $k_4 = hf(x_n + h, y_n + k_3)$	klasyczna metoda Runge-Kutty	$O(h^5)$	$O(h^5)$

4. Zadanie 4

Dane jest równanie różniczkowe. Znaleźć rozwiązanie tego zagadnienia y(x) w przedziale [x0,xk] metodą Eulera oraz metodą Rungego-Kutty.
Równanie jest postaci:

$$y' - kmysin(mx) = k^2msin(mx)cos(mx), \qquad y(x_0) = a$$

Jako x_0 przyjmuję 0, jako x_k przyjmuję , jako m przyjmuję 1, jako k przyjmuję 2.
Zatem równanie przyjmuje postać

$$y' = 4sin(x)cos(x) + 2ysin(x)$$

Z warunkiem brzegowym:

$$y(0) = e^{-2} - 1$$

Dokładne rozwiązanie wynosi:

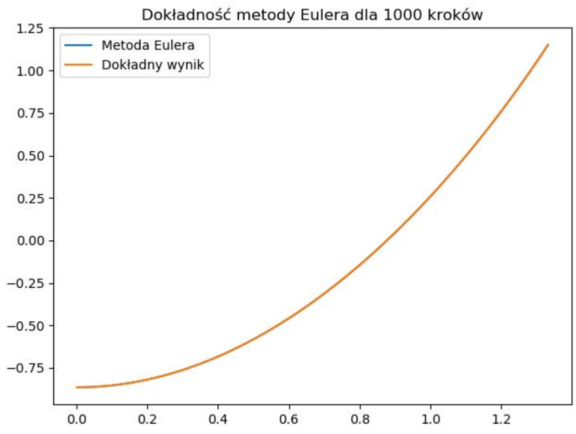
$$y = e^{-2\cos(x)} - 2\cos(x) + 1$$

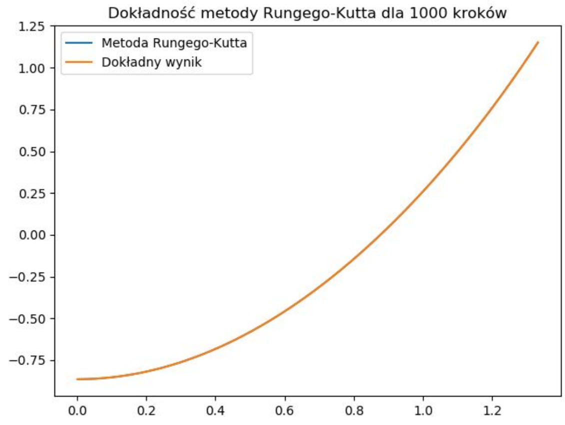
W celu wykonania zadania zmodyfikowane zostały metody:

```cpp
void euler(int iterations) {
	double x0, y;
	x0 = 0;
	y = pow(M_E, -2) - 1;
	double step = (double)2.0 / (double)iterations;
	std::ofstream myfile;
	myfile.open("euler2.txt");
	for (int i = 0; i < iterations; i++) {
		y = y + step * func(x0, y);
		x0 = x0 + step;
		myfile << x0 << std::endl;
		myfile << y << std::endl;
	}
	myfile.close();
}
```

```cpp
void rungeKutta(int iterations) {
	double x0, y;
	x0 = 0;
	y = pow(M_E, -2) - 1;
	double step = (double)2.0 / (double)iterations;
	std::ofstream myfile;
	myfile.open("rungekutta2.txt");
	for (int i = 0; i < iterations; i++) {
		double k1 = step * func(x0, y);
		y = y + step * func(x0+step/2, y+k1/2);
		x0 = x0 + step;
		myfile << x0 << std::endl;
		myfile << y << std::endl;
	}
	myfile.close();
```

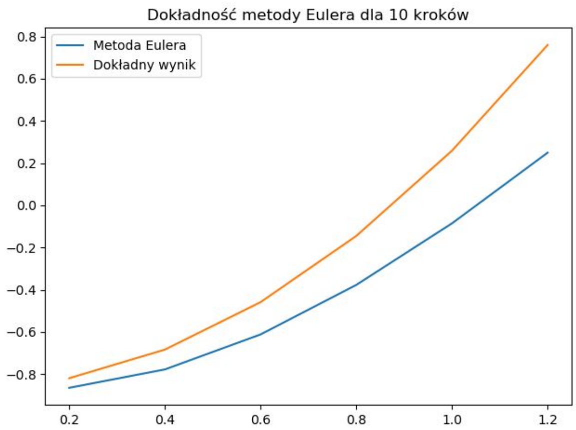
Dla 1000 kroków wykresy wyglądają następująco:

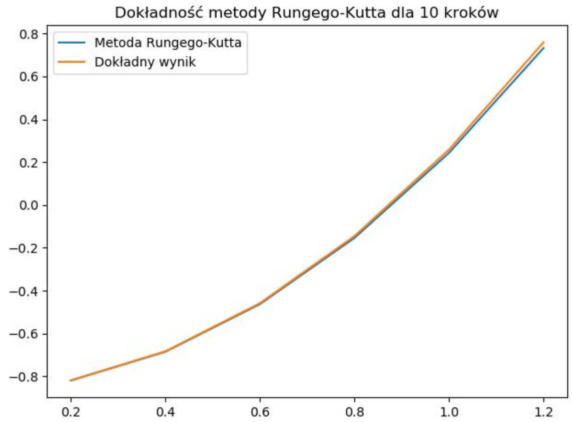




W obu przypadkach nie jesteśmy w stanie odróżnić linii niebieskiej(rozwiązanie testowanej metody) od pomarańczowej (dokładne rozwiązanie).

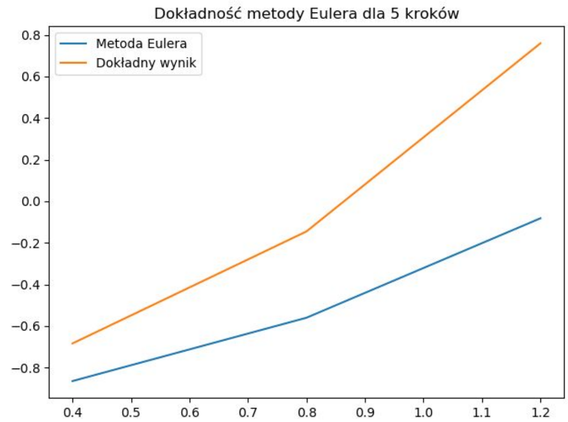
Dla 10 kroków:

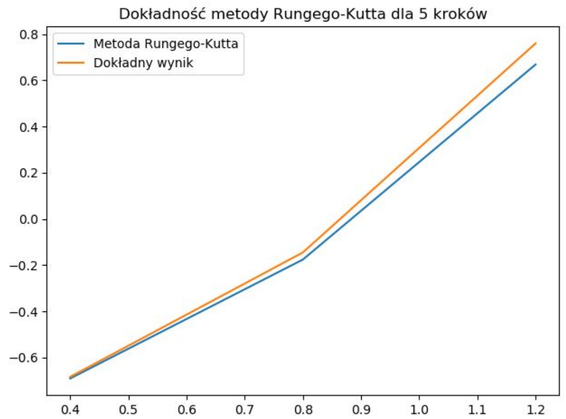




W tym przypadku dostrzegamy już o ile dokładniejsza jest metoda Rungego-Kutta. Podczas gdy dla Rungego-Kutta nie jesteśmy w stanie odróżnić linii dla wartości od 0 do 0.4 to metoda Eulera zupełnie różni się od dokładnego wyniku.

Dla 5 kroków:





Mimo, że zastosowane jest tylko 5 kroków - metoda Rungego-Kutta nadal ma bardzo niski błąd.

Widzimy, że w każdym przypadku metoda Rungego-Kutta przynosi lepsze rezultaty. Zastosowana została metoda Rungego-Kutta 2. rzędu, która ma większy teoretyczny błąd niż metoda Rungego-Kutta 4.rzędu - dla tej metody wyniki powinny być jeszcze lepsze.

W celu obliczania prostych równań różniczkowych warto korzystać z metody Rungego-Kutta 2 lub 4 rzędu. Są one dużo dokładniejsze od metody Rungego-Kutta 1. rzędu (czyli metody Eulera), a nie są dużo trudniejsze w implementacji.