Metody Obliczeniowe w Nauce i Technice

Laboratorium 9 – raport

Mateusz Kocot

Zadanie 2.

Rozpatrywane metody rozwiązywania równań różniczkowych zostały zaimplementowane jako metody w klasie *ODESolver*.

```
template<typename T>
class ODESolver
{
    T (*odeFun)(T vars, double t);

public:
    ODESolver(T odeFun(T vars, double t));

    std::vector<T> euler(double h = 0.001, int n = 50'000, T vars = T(),
                         double t0 = 0.0);
    std::vector<T> backwardEuler(double h = 0.001, int n = 50'000,
                                 T vars = T(), double t0 = 0.0,
                                 int itNum = 100);

    std::vector<T> rungeKuttaSecondOrder(double h = 0.001, int n = 50'000,
                                         T vars = T(), double t0 = 0.0);
    std::vector<T> rungeKuttaFourthOrder(double h = 0.001, int n = 50'000,
                                         T vars = T(), double t0 = 0.0);
};
```

Klasa ta w konstruktorze przyjmuje wskaźnik do funkcji, która charakteryzuje równanie bądź układ równań różniczkowych (*odeFun*). Poniżej przedstawiono implementacje poszczególnych metod. Metody te zostały zaimplementowane w taki sposób, by możliwe było za ich pomocą rozwiązanie układu Lorenza oraz równania z zadania 4.

1. Metoda Eulera

```
template<typename T>
std::vector<T> ODESolver<T>::euler(double h, int n, T vars, double t0)
{
    std::vector<T> result(n + 1);
    result[0] = vars;
    double t = t0;

    for (int i = 1; i <= n; i++)
    {
        result[i] = result[i - 1] + h * odeFun(result[i - 1], t);
        t += h;
    }
    return result;
}
```

2. Metoda niejawna Eulera (ang. „backward Euler method”)

```
template<typename T>
std::vector<T> ODESolver<T>::backwardEuler(double h, int n, T vars, double t0,
                                           int itNum)
{
    std::vector<T> result(n + 1);
    result[0] = vars;
    double t = t0;

    for (int i = 1; i <= n; i++)
    {
        t += h;
        result[i] = result[i - 1];
        for (int j = 0; j < itNum; j++)
        {
            result[i] = result[i - 1] + h * odeFun(result[i], t);
        }
    }
    return result;
}
```

(3). Nie zaimplementowano metody Rungego-Kutty 1. rzędu, gdyż jest ona równoważna metodzie Eulera.

3. Metoda Rungego-Kutty 2. rzędu (metoda punktu pośredniego)

```
template<typename T>
std::vector<T> ODESolver<T>::rungeKuttaSecondOrder(double h, int n, T vars,
                                                   double t0)
{
    std::vector<T> result(n + 1);
    result[0] = vars;
    double t = t0;

    for (int i = 1; i <= n; i++)
    {
        result[i] = result[i - 1] + h * odeFun(result[i - 1] + h / 2.0 *
                    odeFun(result[i - 1], t), t + h / 2.0);
        t += h;
    }
    return result;
}
```

4. Metoda Rungego-Kutty 4. rzędu

```
template<typename T>
std::vector<T> ODESolver<T>::rungeKuttaFourthOrder(double h, int n, T vars,
                                                   double t0)
{
    std::vector<T> result(n + 1);
    result[0] = vars;
    double t = t0;

    for (int i = 1; i <= n; i++)
    {
        auto k1 = odeFun(result[i - 1], t);
        auto k2 = odeFun(result[i - 1] + h / 2.0 * k1, t + h / 3.0);
        auto k3 = odeFun(result[i - 1] + h / 2.0 * k2, t + 2 * h / 3.0);
        auto k4 = odeFun(result[i - 1] + h * k3, t + h);
        result[i] = result[i - 1] + h / 6.0 * (k1 + 2 * k2 + 2 * k3 + k4);
        t += h;
    }
    return result;
}
```

Powyższe metody wykorzystano do obliczenia układu Lorenza:

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

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

Jako punkt startowy przyjęto $x_0 = 0$, $y_0 = 2$ i $z_0 = 20$.

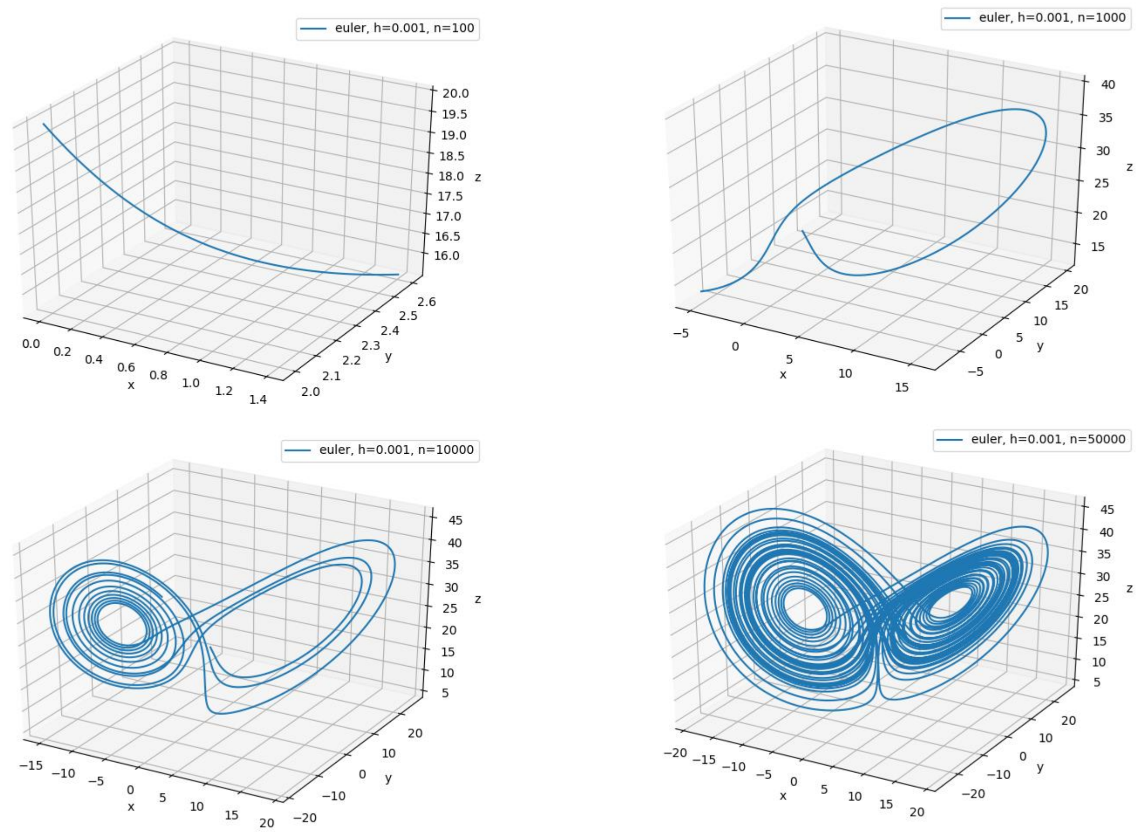
Zaimplementowano funkcję *LorenzFuns*, która zostanie wykorzystana do utworzenia obiektu klasy *ODESolver*.

```
Triple lorenzFuns(Triple vars, double t)
{
    double x = vars.getX();
    double y = vars.getY();
    double z = vars.getZ();

    double f_x = sigma * (y - x);
    double f_y = x * (ro - z) - y;
    double f_z = x * y - beta * z;

    return Triple(f_x, f_y, f_z);
}
```

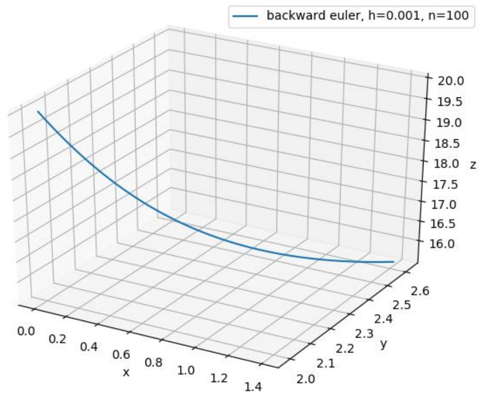
Klasa *Triple* została przeze mnie zaimplementowana w celu wygodnego przetrzymywania i operowania na współrzędnych x, y, z.

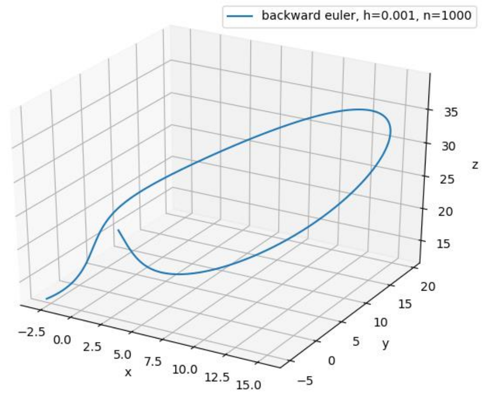
We wszystkich metodach użyto kroku $t_{n+1} - t_n = h = 0.001$. Poniżej przedstawione zostaną wykresy rozwiązań układu dla 100, 1000, $10\,000$ i $50\,000$ kroków. Wykresy sporządzono w języku Python z wykorzystaniem pakietu Matplotlib.

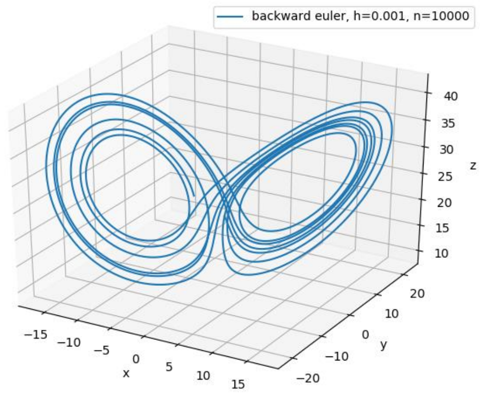
1. Metoda Eulera

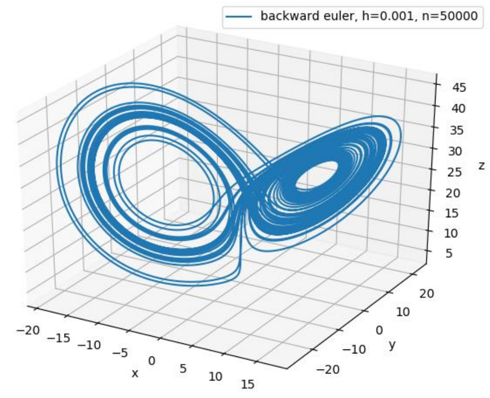


2. Metoda niejawna Eulera

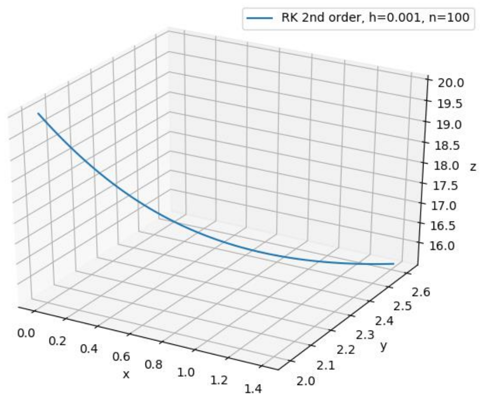


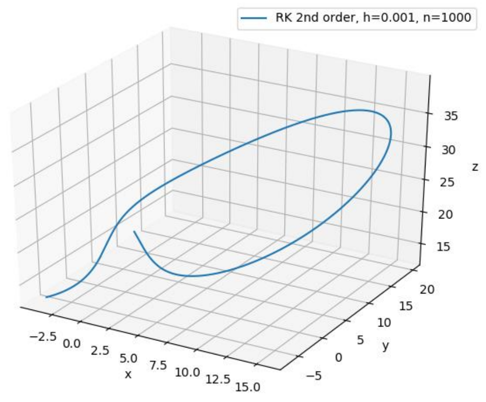


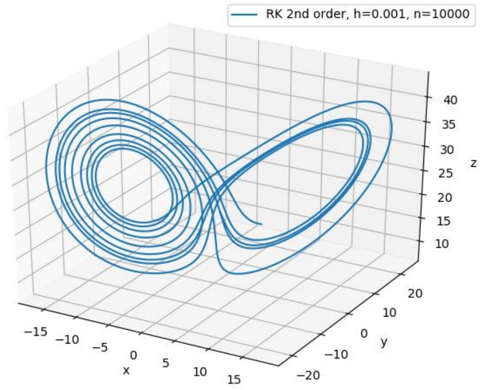


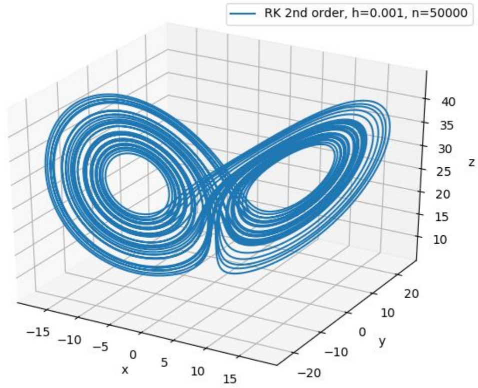


3. Metoda Rungego-Kutty 2. rzędu

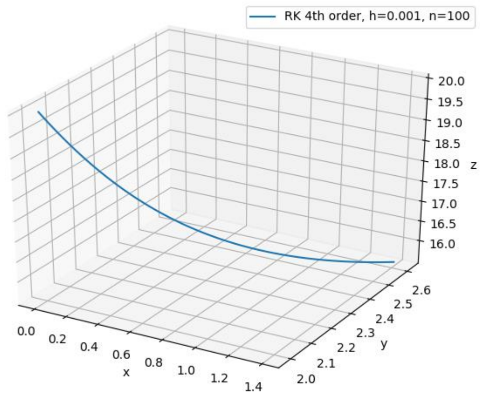


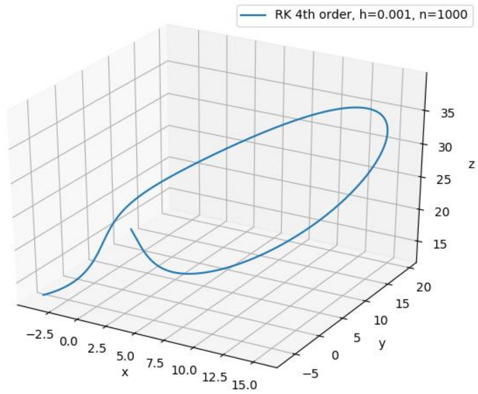


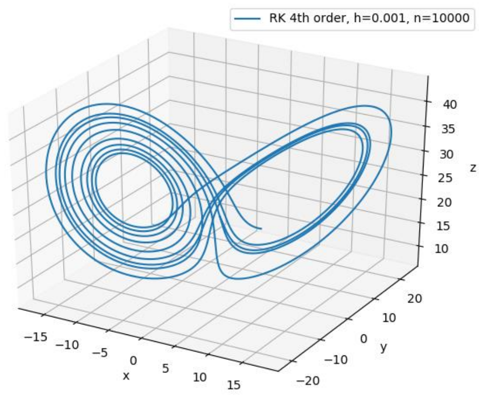


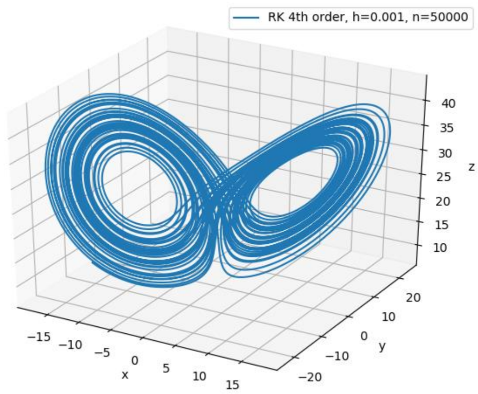


4. Metoda Rungego-Kutty 4. rzędu









Jak widać, wszystkie metody zostały zaimplementowane poprawnie. Różne wyglądy atraktorów na wykresach spowodowane są oczywiście różnicami w metodach.

Zadanie 3.

Najprostszą z rozpatrywanych metod jest metoda Eulera.

Rozwiązywać będziemy równanie postaci:

$$\frac{dy}{dt} = f(t, y), \;\; y(t_0) = y_0$$

Rozwiązanie przybliżane będzie w punktach $t_0, t_1, \ldots$. Oznaczmy $t_{n+1} - t_n = h$. Heurystyka metody Eulera polega na przybliżeniu funkcji w punkcie t_{n+1} za pomocą prostej stycznej do funkcji w punkcie poprzednim (t_n):

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

Znając więc wartość w punkcie t_0, możemy szacować wartości w kolejnych punktach t_n.

Metoda niejawna Eulera (ang. „backward Euler method”) jest podobna do zwykłej metody Eulera. Metoda ta przybliża funkcje w punkcie t_{n+1} za pomocą prostej stycznej do funkcji także w tym punkcie według wzoru:

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

Tym razem y_{n+1} występuje po obu stronach równania. Można zastosować metodę punktu stałego (ang. „fixed-point iteration”):

$$y_{n+1}^{[0]} = y_n$$

$$y_{n+1}^{[i+1]} = y_n + h \cdot f\left(t_{n+1}, y_{n+1}^{[i]}\right)$$

Metodę Rungego-Kutty 2. rzędu także można postrzegać jako zmodyfikowaną metodę Eulera. Przybliżenia funkcji w kolejnych punktach obliczane są według wzoru:

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

Metoda ta jest nazywana metodą punktu pośredniego, gdyż brana pod uwagę jest wartość funkcji f w punkcie $t_n + \frac{h}{2}$.

Najbardziej rozbudowaną i najczęściej stosowaną jest metoda Rungego-Kutty 4. rzędu. Wzór na kolejne przybliżenia wygląda tu następująco:

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

k_1, podobnie jak w metodzie Eulera, wyraża nachylenie funkcji na początku przedziału.

$$k_1 = f(t_n, y_n)$$

k_2 obliczane jest z wykorzystaniem k_1 i określa nachylenie w punkcie środkowym przedziału.

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right)$$

k_3 także określa nachylenie w punkcie środkowym. Tym razem jednak, wykorzystywane jest k_2.

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right)$$

Ostatecznie, k_4 wyraża nachylenie funkcji na końcu przedziału. Do obliczenia wykorzystywane jest k_2.

$$k_4 = f(t_n + h, y_n + hk_3)$$

W ten właśnie sposób jesteśmy w stanie dość dokładnie przybliżyć rozwiązania równań różniczkowych bez wyznaczania pochodnych funkcji f.

Zadanie 4.

Tym razem, rozpatrywane jest równanie:

$$\frac{dy}{dx} - kmy\sin(mx) = k^2 m \sin(mx)\cos(mx)$$

Przyjęto: $m = 1$, $k = 2$. Rozwiązanie będzie wyznaczane w przedziale $[x_0, x_k] = [0, 40]$.

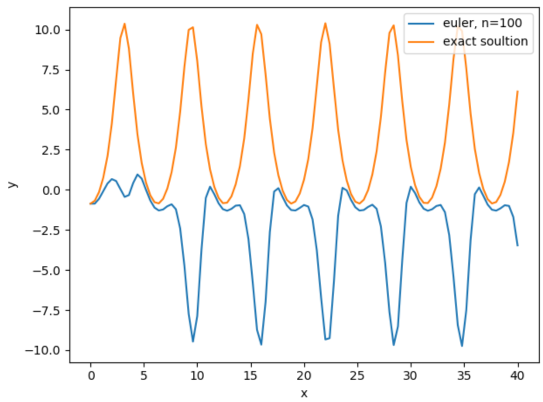
Do skonstruowania obiektu klasy *ODESolver* wykorzystana zostanie funkcja *task4Fun*:

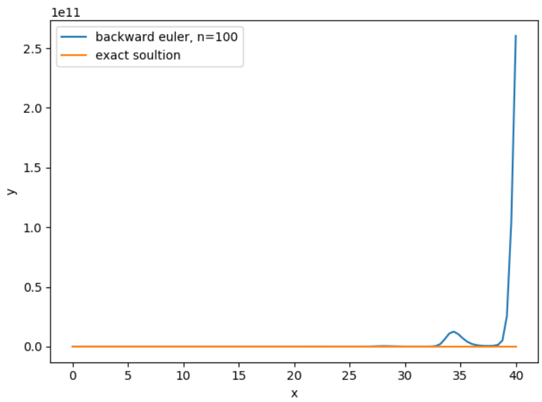
```
double task4Fun(double y, double x)
{
    return k * m * y * std::sin(m * x) +
           k * k * m * std::sin(m * x) * std::cos(m * x);
}
```

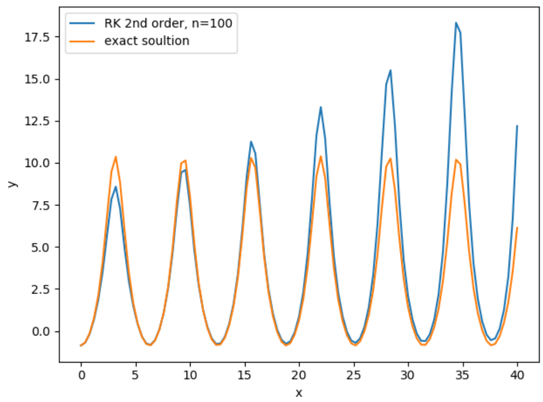
Na wykresach poniżej przedstawiono wyniki działania rozpatrywanych metod odpowiednio dla $n = 100, 500, 1000$ i $10\,000$ kroków. Na wykresach przedstawiono także prawdziwe rozwiązanie:

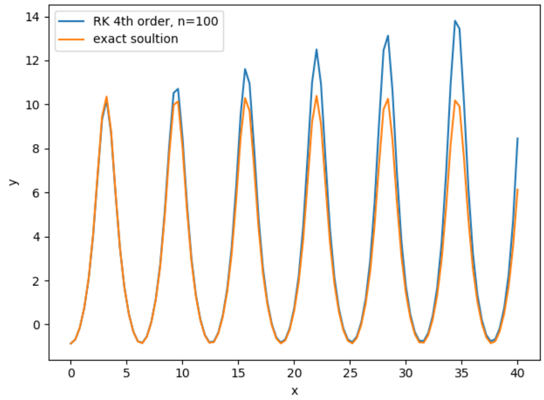
$$y(x) = e^{-k\cos(mx)} - k\cos(mx) + 1$$

1. $n = 100$

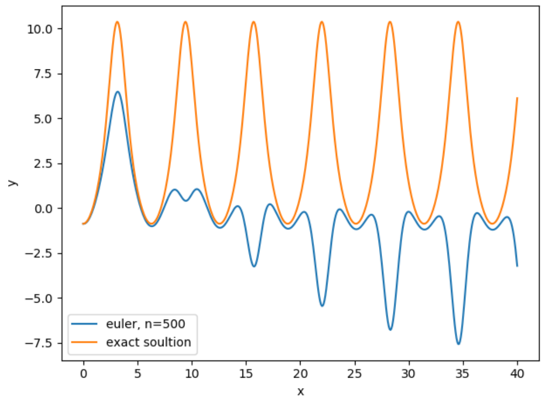


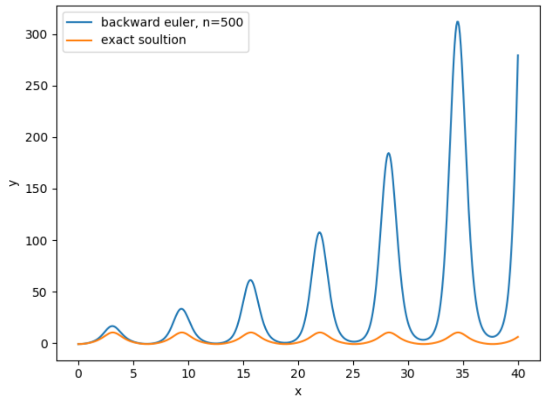


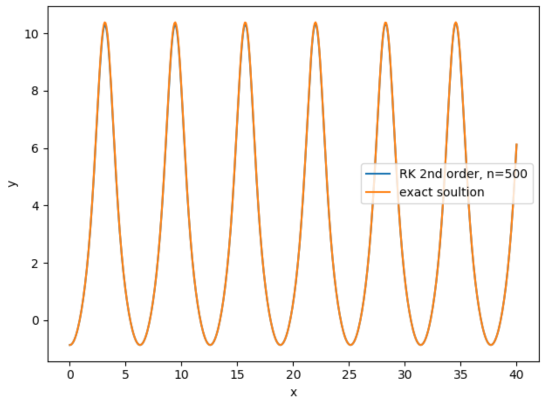

RK 2nd order, n=100
exact soultion
x
y

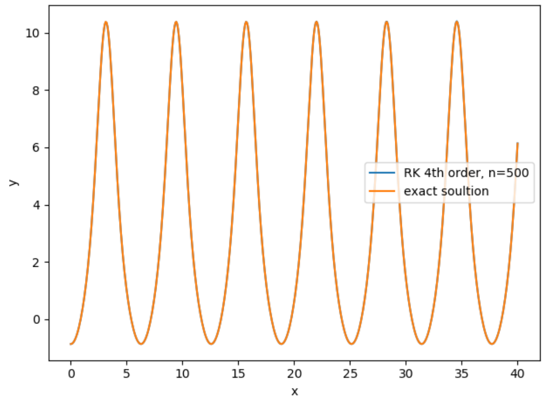

RK 4th order, n=100
exact soultion
x
y

2. $n = 500$

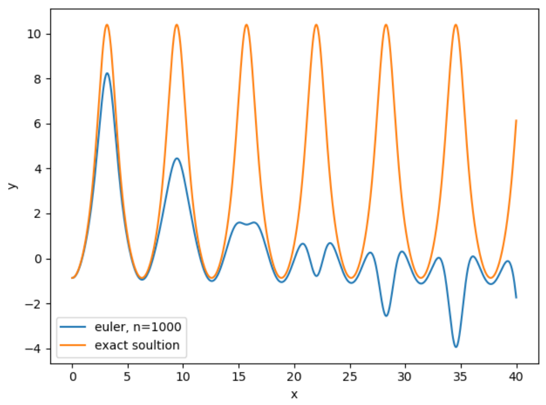

euler, n=500
exact soultion
x
y

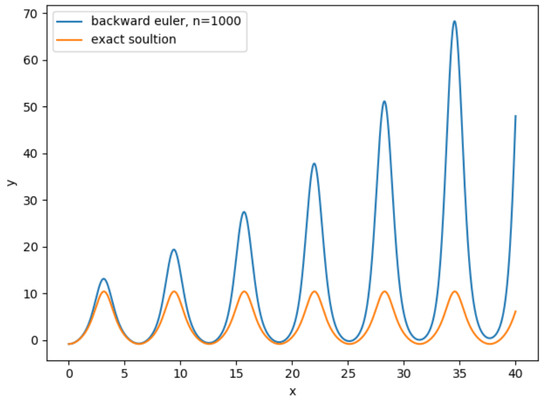

backward euler, n=500
exact soultion
x
y

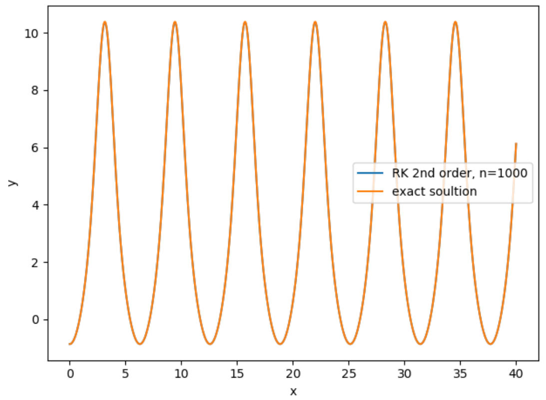

RK 2nd order, n=500
exact soultion
x
y

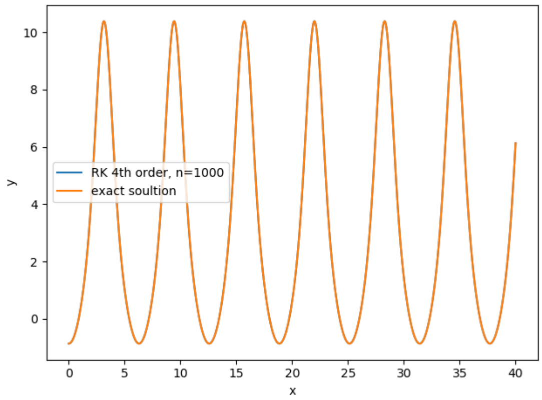

RK 4th order, n=500
exact soultion
x
y

3. $n = 1000$

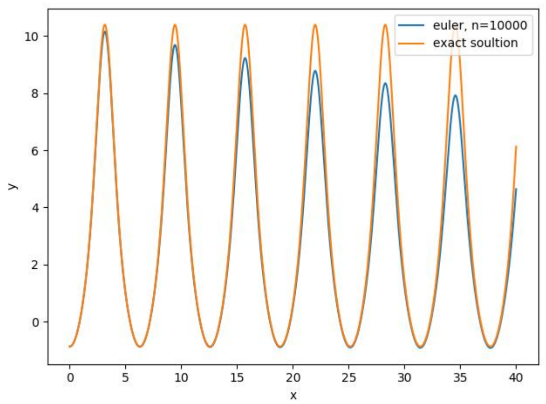


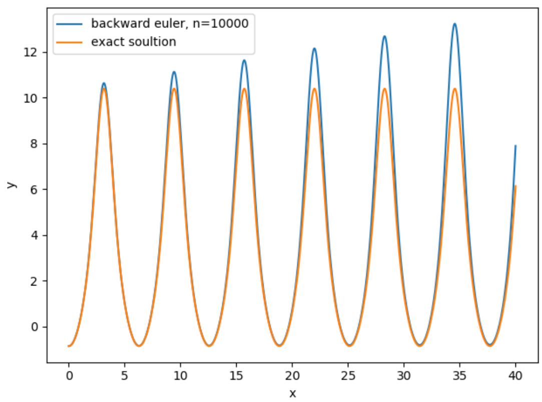


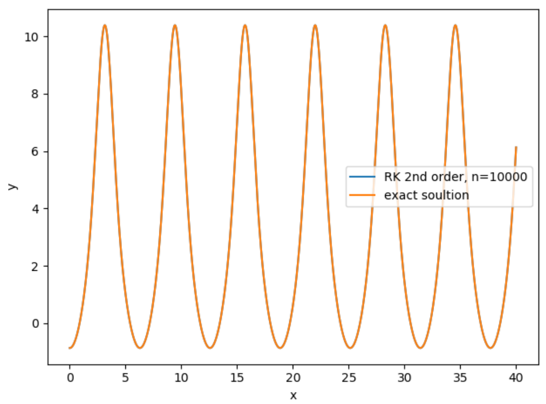


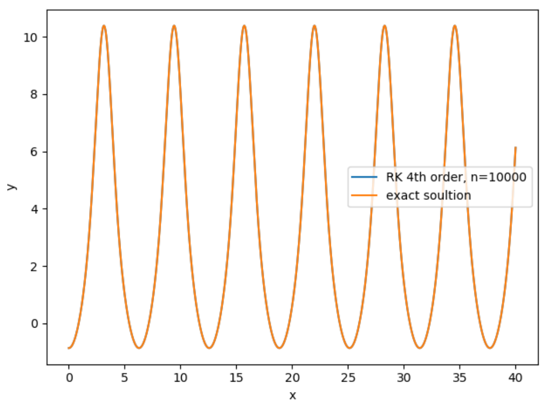


4. $n = 10\,000$









Metody Eulera nawet dla $n = 10\,000$ nie są dokładne. Metody Rungego-Kutty natomiast już dla $n = 500$ przybliżają rozwiązanie bardzo dokładnie. Dla $n = 100$ można zauważyć, że metoda Rungego-Kutty 4. rzędu przybliża rozwiązanie lepiej od metody 2. rzędu. Warto zaznaczyć, że funkcja na rozpatrywanym przedziale jest mocno zmienna. W związku z tym, można było łatwo przewidzieć, że metody Rungego-Kutty będą dużo dokładniejsze od metod Eulera.