Julia Plewa
MOwNiT
Laboratorium 9

1. **Proszę na podstawie informacji nabytych w zadaniu 5 poprzedniego laboratorium zaimplementować algorytm Verleta (leap-frog). Może być to zrealizowane w postaci zwyczajnej funkcji, która przyjmuje stosowne parametry.**

kod:

```
std::vector<std::vector<double>> leapFrogAlgorithm(std::vector<double> a, double x0,
    double v0, double start, double dt, int N)
{
    std::vector<std::vector<double>> results;
    std::vector<double> t(N);
    std::vector<double> x(N);
    std::vector<double> v(N);

    t[0] = start;
    x[0] = x0;
    v[0] = v0;

    results.push_back({t[0], x[0], v[0], a[0]});

    for (int i = 0; i < N - 1; i++)
    {
        t[i + 1] = t[i] + dt;
        x[i + 1] = x[i] + v[i] * dt + dt * dt * a[i] / 2;
        v[i + 1] = v[i] + dt * (a[i] + a[i + 1]) / 2;
        results.push_back({t[i + 1], x[i + 1], v[i + 1], a[i + 1]});
    }
    return results;
}
```

2. **Proszę z zamieszczonego w treści zadań pdf-a wybrać dowolne 4 z 6 zadań dot. symulacji komputerowych prostych układów fizycznych. Wykorzystując implementacje z Lab8 proszę rozwiązać te problemy oraz przedstawić STOSOWNE opracowanie.**

Do zrealizowania zadania konieczne było wprowadzenie modyfikacji w kodzie z poprzednich zajęć (wprowadzenie abstrakcji dla równań różniczkowych).

Euler:

```
class Euler : IDifferentialEquationSolver
{
    public:
    Euler(IDifferentialEquation* differentialEquation,
        std::vector<double> initialState) :
        IDifferentialEquationSolver(differentialEquation, initialState) {}

    std::vector<std::vector<double>> solve(double t0, double h, int n)
    {
        std::vector<std::vector<double>> results;
        double t = t0;
        std::vector<double> state = initialState;
        state.push_back(t);

        for (int j = 0; j < n; j++)
        {
            std::vector<double> derivatives = differentialEquation ->
                getDerivatives(state);
            for (int i = 0; i < derivatives.size(); i++)
            {
                state[i] = state[i] + h * derivatives[i];
            }
            t += h;
            state[state.size() - 1] = t;
            results.push_back(state);
        }
        return results;
    }
};
```

Backward Euler:

```
class BackwardEuler : IDifferentialEquationSolver
{
    public:
    BackwardEuler(IDifferentialEquation* differentialEquation,
        std::vector<double> initialState) :
        IDifferentialEquationSolver(differentialEquation, initialState) {}

    std::vector<std::vector<double>> solve(double t0, double h, int n)
    {
        std::vector<std::vector<double>> results;

        double t = t0;
        std::vector<double> state = initialState;
        state.push_back(t);

        for (int j = 0; j < n; j++)
        {
            t += h;
            std::vector<double> derivatives = ApproximateDerivatives(state, h);
            for (int i = 0; i < derivatives.size(); i++)
            {
                state[i] = derivatives[i];
            }
            state[state.size() - 1] = t;
            results.push_back(state);
        }
        return results;
    }
    private:
    std::vector<double> ApproximateDerivatives(std::vector<double> state, double h)
    {
        std::vector<double> state0(state.size() - 1);
        std::vector<double> state1(state.size() - 1);

        std::vector<double> derivatives = differentialEquation ->
            getDerivatives(state);
        for (int i = 0; i < state0.size(); i++)
        {
            state0[i] = state[i] + h * derivatives[i];
        }
        derivatives = differentialEquation -> getDerivatives(state0);
```

```
        for (int i = 0; i < state1.size(); i++)
        {
            state1[i] = state[i] + h * derivatives[i];
        }
        for (int j = 0; j < FIXED_POINT - 2; j++)
        {
            derivatives = differentialEquation -> getDerivatives(state1);
            for (int i = 0; i < state0.size(); i++)
            {
                state0[i] = state1[i];
            }
            for (int i = 0; i < state1.size(); i++)
            {
                state1[i] = state[i] + h * derivatives[i];
            }
        }
        return state1;
    }
};
```

Runge-Kutta 2:

```
class RungeKutta2 : IDifferentialEquationSolver
{
    public:
    RungeKutta2(IDifferentialEquation* differentialEquation,
        std::vector<double> initialState) :
        IDifferentialEquationSolver(differentialEquation, initialState) {}

    std::vector<std::vector<double>> solve(double t0, double h, int n)
    {
        std::vector<std::vector<double>> results;
        double t = t0;
        std::vector<double> state = initialState;
        state.push_back(t);

        for (int j = 0; j < n; j++)
        {
            std::vector<double> derivatives = differentialEquation ->
                getDerivatives(state);
            std::vector<double> state1(derivatives.size());
            for (int i = 0; i < state1.size(); i++)
```

```
            {
                state1[i] = state[i] + h * derivatives[i] / 2;
            }
            derivatives = differentialEquation -> getDerivatives(state1);
            for (int i = 0; i < state1.size(); i++)
            {
                state[i] += h * derivatives[i];
            }
            t += h;
            state[state.size() - 1] = t;
            results.push_back(state);
        }
        return results;
    }
};
```

Runge-Kutta 4:

```
class RungeKutta4 : IDifferentialEquationSolver
{
    public:
    RungeKutta4(IDifferentialEquation* differentialEquation,
        std::vector<double> initialState) :
        IDifferentialEquationSolver(differentialEquation, initialState) {}

    std::vector<std::vector<double>> solve(double t0, double h, int n)
    {
        std::vector<std::vector<double>> results;
        double t = t0;

        std::vector<double> state = initialState;
        state.push_back(t);

        for (int j = 0; j < n; j++)
        {
            std::vector<double> k(initialState.size());
            std::vector<double> sum(initialState.size());
            std::vector<double> variables(initialState.size());

            // k1
            std::vector<double> derivatives = differentialEquation ->
                getDerivatives(state);
```

```
            for (int i = 0; i < derivatives.size(); i++)
            {
                k[i] = h * derivatives[i];
                sum[i] = k[i];
            }

            // k2, k3
            for (int l = 0; l < 2; l++)
            {
                for (int i = 0; i < derivatives.size(); i++)
                {
                    variables[i] = state[i]  + k[i] / 2;
                }
                derivatives = differentialEquation -> getDerivatives(variables);
                for (int i = 0; i < derivatives.size(); i++)
                {
                    k[i] = h * derivatives[i];
                    sum[i] += 2 * k[i];
                }
            }
            // k4
            for (int i = 0; i < derivatives.size(); i++)
            {
                variables[i] = state[i]  + k[i];
            }
            derivatives = differentialEquation -> getDerivatives(variables);
            for (int i = 0; i < derivatives.size(); i++)
            {
                k[i] = h * derivatives[i];
                sum[i] += k[i];
            }
            for (int i = 0; i < derivatives.size(); i++)
            {
                state[i] += sum[i] / 6;
            }
            t += h;
            state[state.size() - 1] = t;
            results.push_back(state);
        }
        return results;
    }
};
```

Boost:

```
class BoostSolver : IDifferentialEquationSolver
{
    public:
    BoostSolver(IDifferentialEquation* differentialEquation,
        std::vector<double> initialState) :
        IDifferentialEquationSolver(differentialEquation, 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> state(differentialEquation -> getDimensions());
            for (int i = 0; i < state.size(); i++)
            {
                state[i] = x[i];
            }
            std::vector<double> derivatives = this -> differentialEquation ->
                getDerivatives(state);
            for (int i = 0; i < derivatives.size(); i++)
            {
                dxdt[i] = derivatives[i];
            }
        };

        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)
    {
        std::vector<double> result(differentialEquation -> getDimensions());
        for (int i = 0; i < result.size(); i++)
```

```
        {
            result[i] = x[i];
        }
        result.push_back(t);
        solution.push_back(result);
    }

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

a. (zad1) Wahadło matematyczne

Wahadło matematyczne jest punktem materialnym poruszającym się po okręgu w pionie w jednorodnym polu grawitacyjnym. Jest opisane równaniem:

$$\frac{\partial^2 x}{dt^2} + \frac{g}{l} sin\theta = 0\,,$$

gdzie θ to kąt wychylenia, g to przyspieszenie ziemskie, a l to długość nici.

Dla niewielkiej amplitudy rozwiązanie równania przybliżane jest dla $\theta \approx sin\,\theta$. Uzyskujemy wtedy równanie drgań harmonicznych:

$$\theta(t) = \theta_0 sin(\omega t + \varphi)\,,$$

gdzie θ_0 to amplituda drgań, $\omega = \sqrt{\frac{g}{l}}$ to częstość kołowa drgań, a φ to faza początkowa drgań.

Rozwiązanie dokładne ma postać:

$$\frac{\partial \theta}{\partial t} = \omega\,,$$
$$\frac{\partial \theta}{\partial t} = -\frac{g}{l} sin\theta\,.$$

Kod:

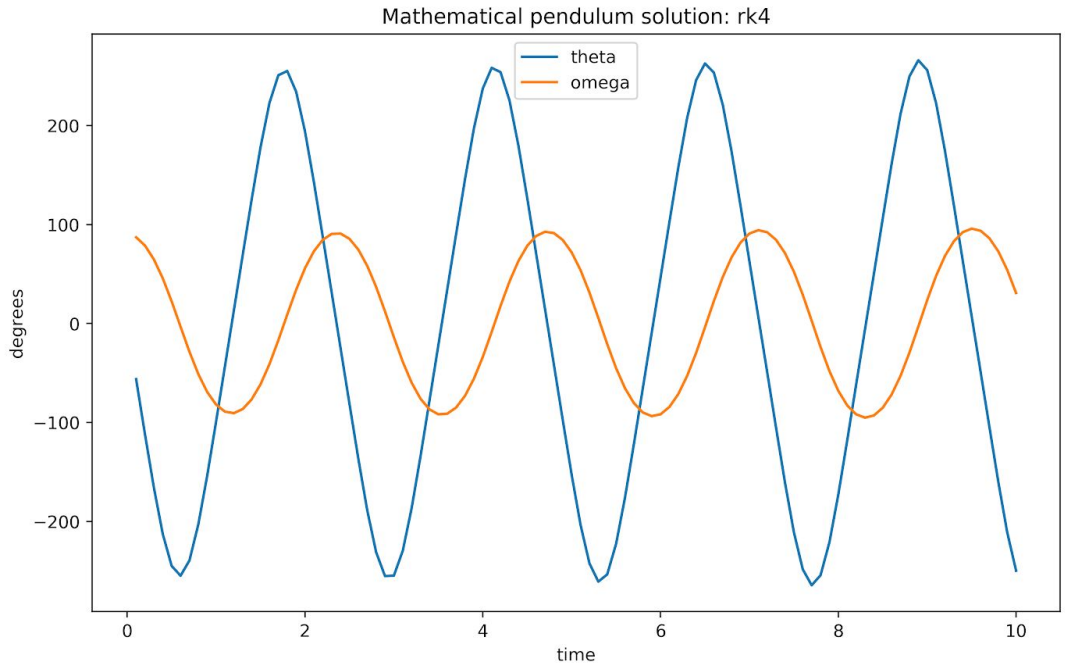
```
class MathematicalPendulumODE : public IDifferentialEquation
{
    public:
    MathematicalPendulumODE(std::vector<double> constants) :
        IDifferentialEquation(constants) {}

    std::vector<double> getDerivatives(std::vector<double> variables)
    {
        double omega = variables[0];
        double theta = variables[1];

        std::vector<double> derivatives({
            -9.80665 * sin(theta) / getL(),
            omega
        });
        return derivatives;
    }

    int getDimensions()
    {
        return 2;
    }

    private:
    double getL()
    {
        return constants[0];
    }
};
```

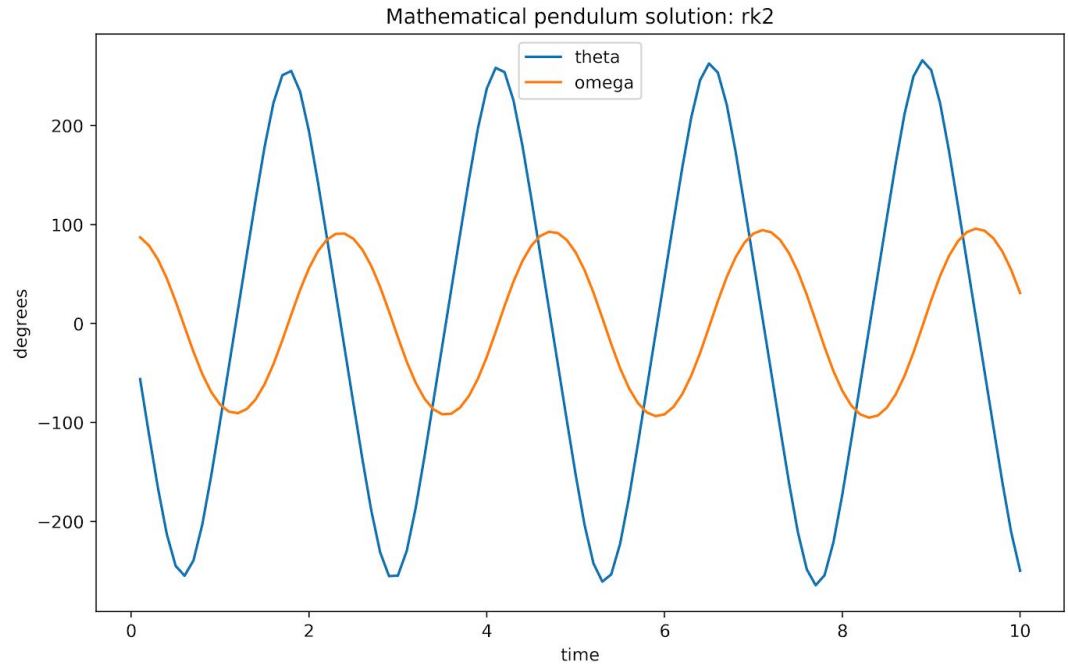
Wyniki:

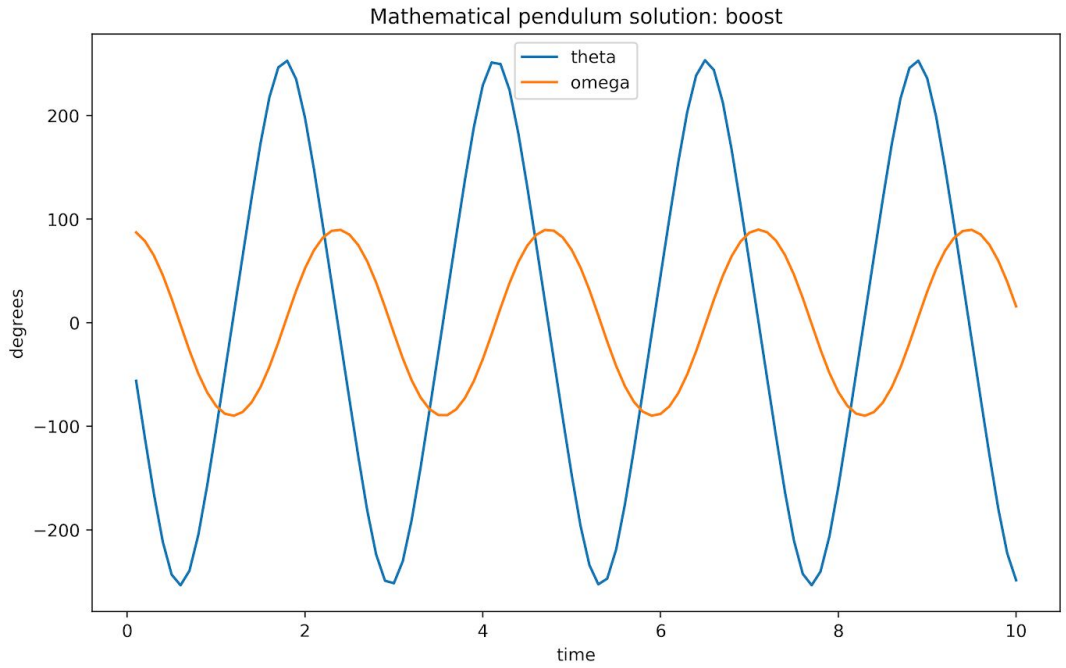
Przypadek 1

$\theta_0 = \frac{\pi}{2} = 90^\circ$, $l = 1$

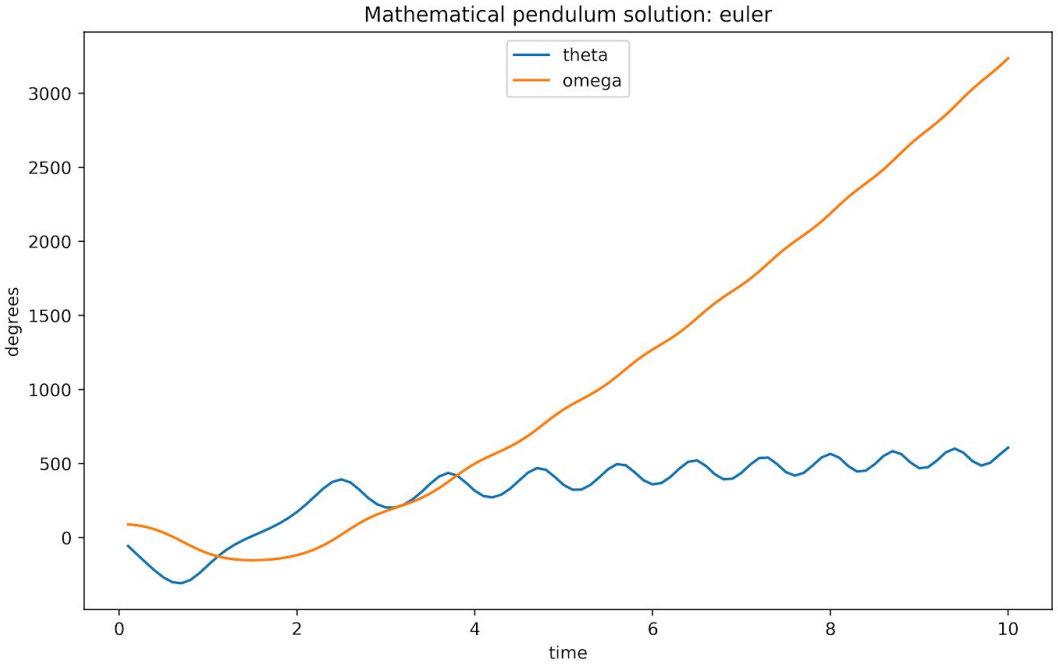
Metody dające rozwiązania zgodne z oczekiwaniami: RK2, RK4, Boost

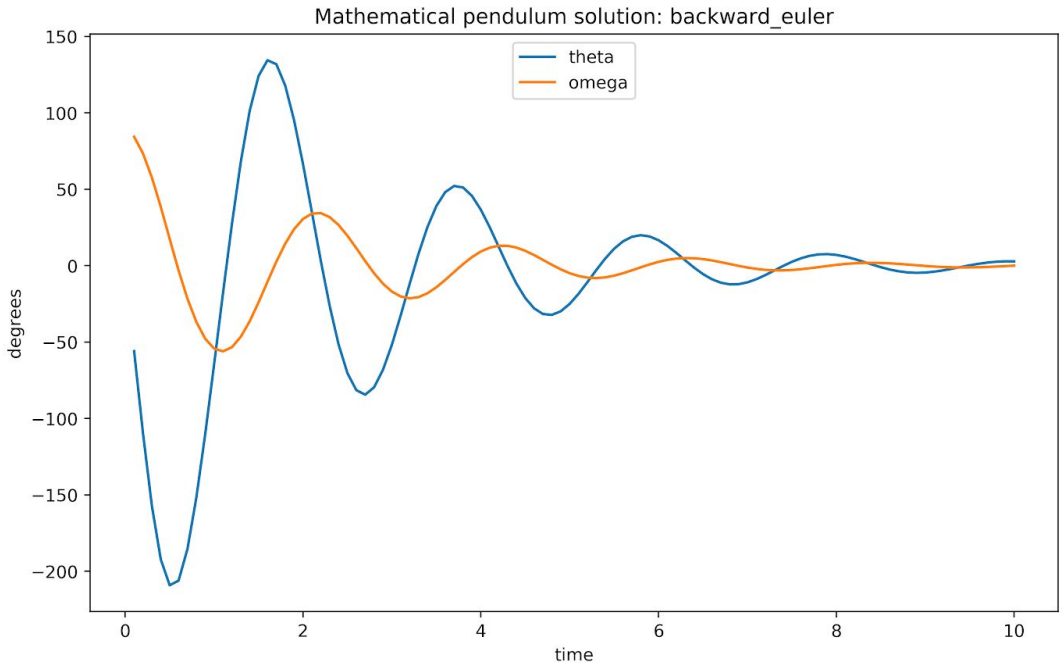






Metody dające rozwiązania niezgodne z oczekiwaniami: Euler, Backward Euler

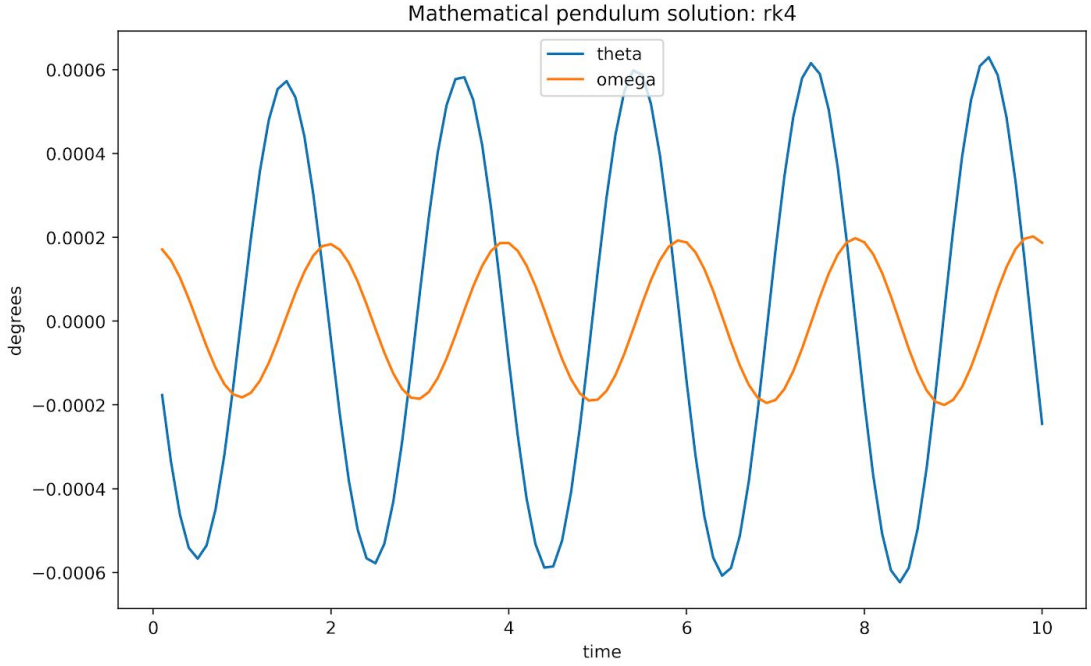




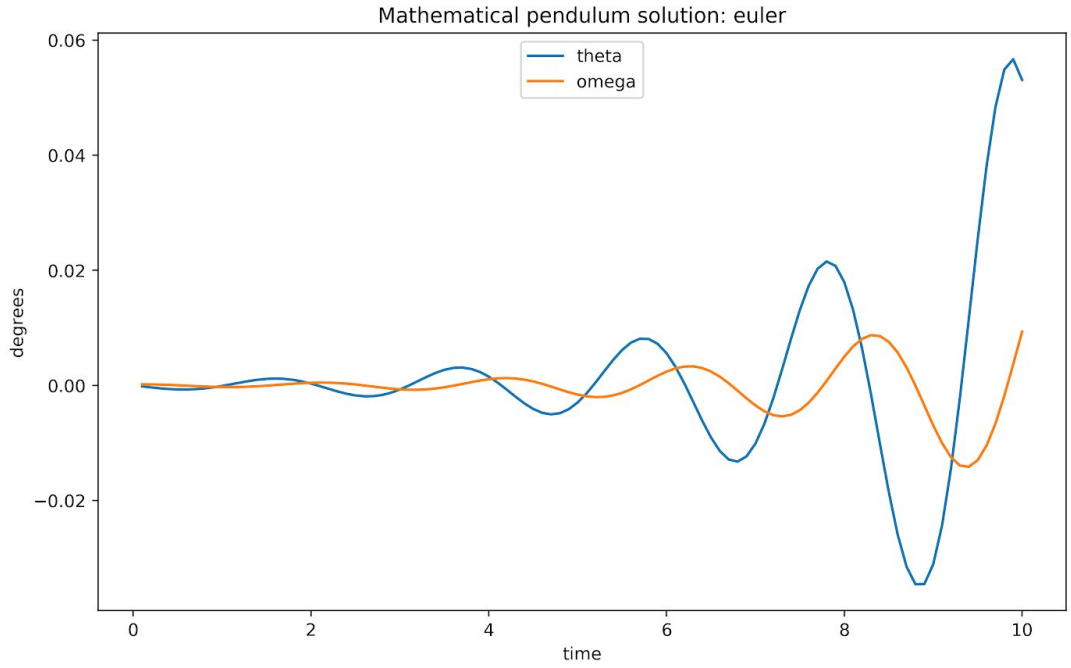
Przypadek 2

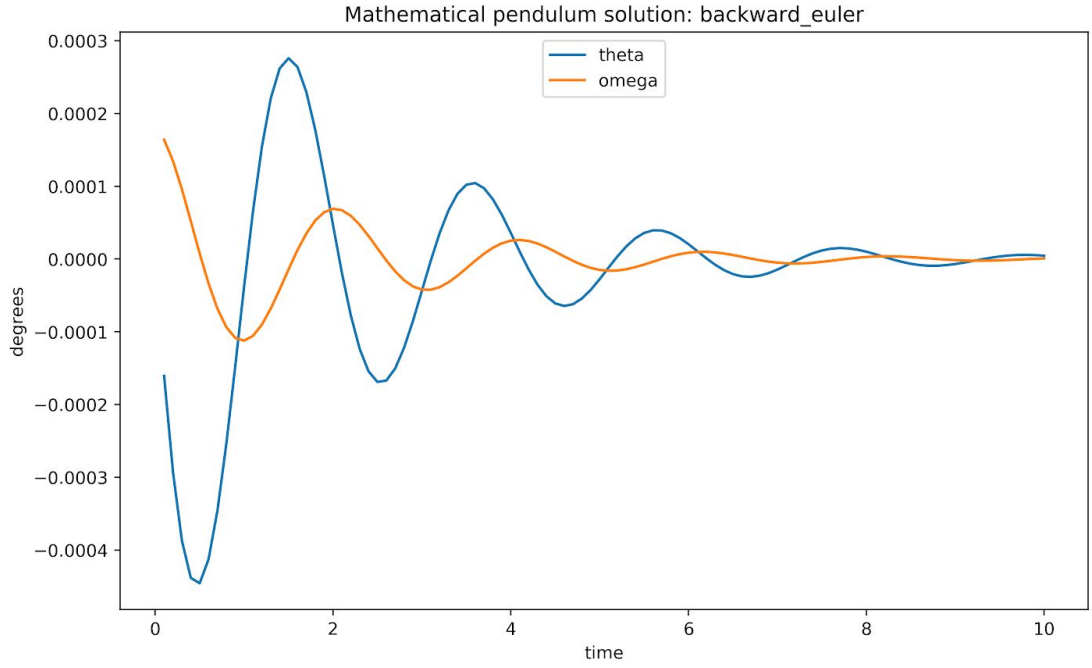
$\theta_0 = \frac{\pi}{1000000} = 0.00018^\circ$, $l = 1$

Metody dające rozwiązania zgodne z oczekiwaniami: RK2, RK4, Boost



Metody dające rozwiązania niezgodne z oczekiwaniami: Euler, Backward Euler





Wnioski:

Metoda Eulera (w wariancie explicit i implicit) jest obarczona zbyt dużym błędem, aby poprawnie modelować zadany układ. Dobrze sprawdziłaby się tutaj również wspomniany algorytm skokowy. Pozostałe zaimplementowane metody prowadzą do poprawnych rozwiązań nawet dla wahadła o większej długości.

b. (zad3) Model drapieżnik-ofiara

Równanie Lotki-Volterry (model drapieżnik-ofiara) umożliwia modelowanie układów dynamicznych występujących w ekosystemach, na przykład symulacji zmian w populacji drapieżników i ofiar. Układ równań ma postać:

$$\frac{\partial x}{\partial t} = -a \cdot x + c \cdot d \cdot x \cdot y,$$
$$\frac{\partial y}{\partial t} = b \cdot y - d \cdot x \cdot y,$$

gdzie a to współczynnik śmierci drapieżników z powodu braku ofiar, b to współczynnik narodzin ofiar, gdy nie ma drapieżników, c to efektywność, z jaką drapieżnik wykorzystuje energię pozyskaną ze zjedzenia ofiar, a d to efektywność uśmiercania ofiar przez drapieżników

Kod:

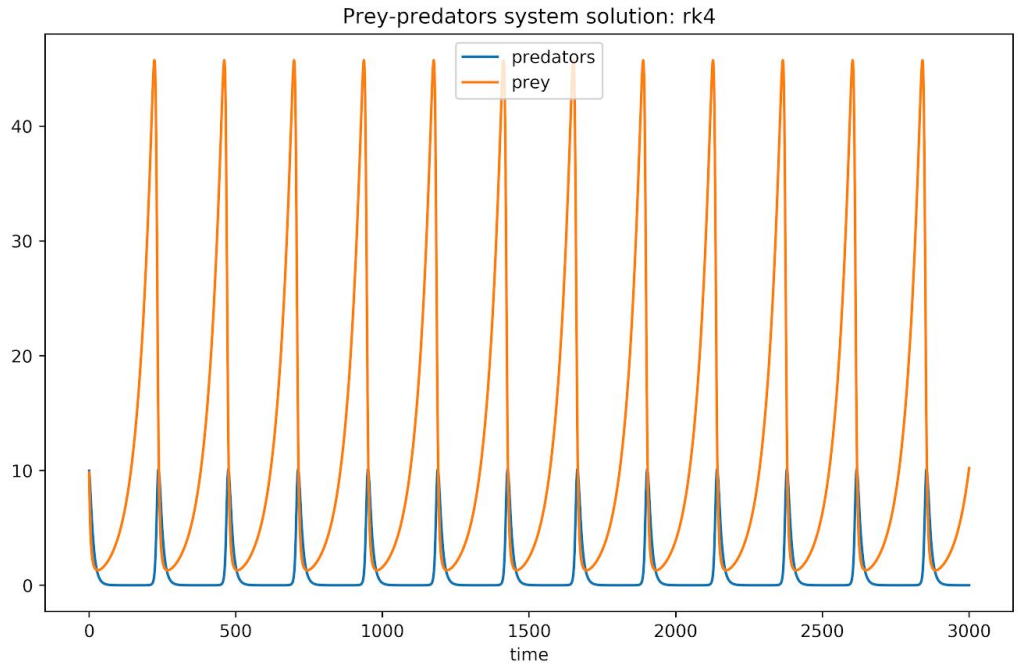
```
class PredatorPreyODE : public IDifferentialEquation
{
    public:
    PredatorPreyODE(std::vector<double> constants) :
        IDifferentialEquation(constants) {}

    std::vector<double> getDerivatives(std::vector<double> variables)
    {
        double x = variables[0];
        double y = variables[1];

        std::vector<double> derivatives({
            getC() * getD() * x * y - getA() * x,
            getB() * y - getD() * x * y
        });

        return derivatives;
    }

    int getDimensions()
    {
        return 2;
    }

    private:
```

```
    double getA()
    {
        return constants[0];
    }
    double getB()
    {
        return constants[1];
    }
    double getC()
    {
        return constants[2];
    }
    double getD()
    {
        return constants[3];
    }
};
```

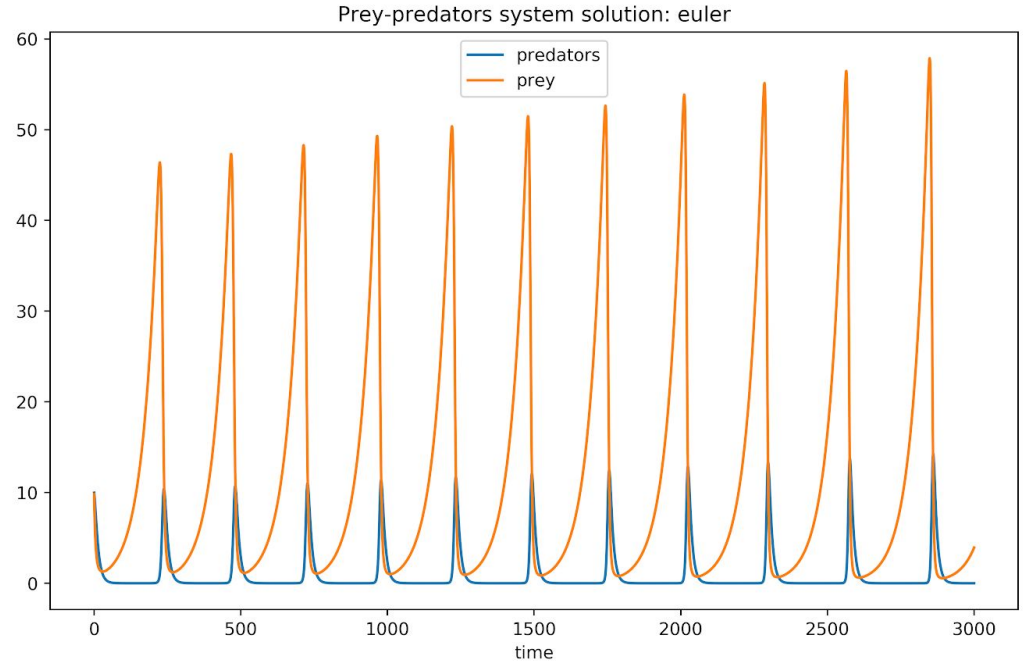
Wyniki:

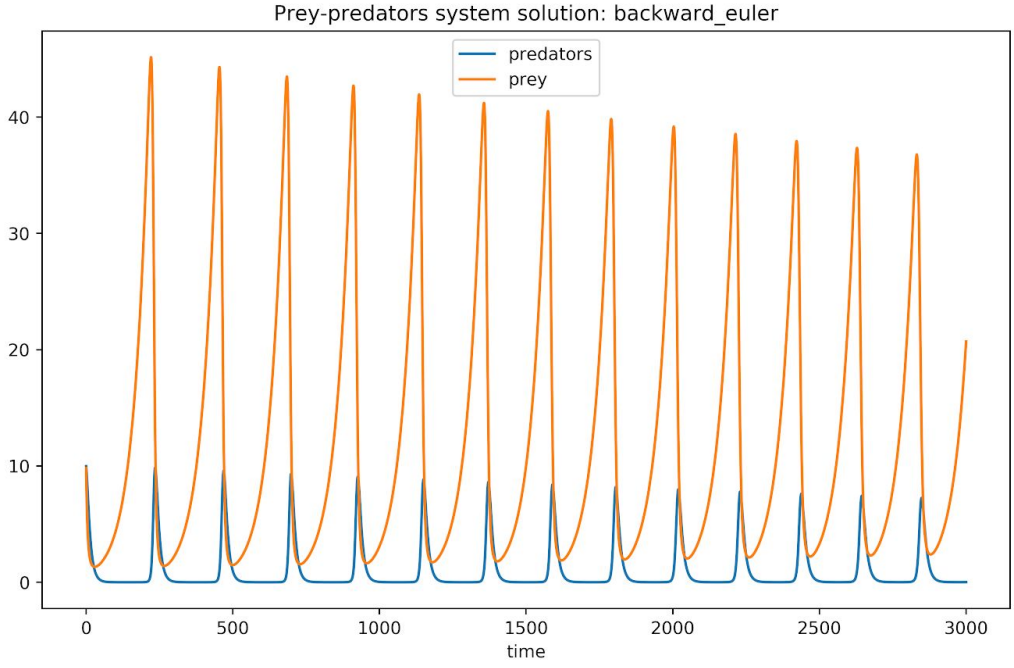
$a = 0.1,\ b = 0.02,\ c = 0.4,\ d = 0.02$

Metody dające rozwiązania zgodne z oczekiwaniami: RK2, RK4, Boost

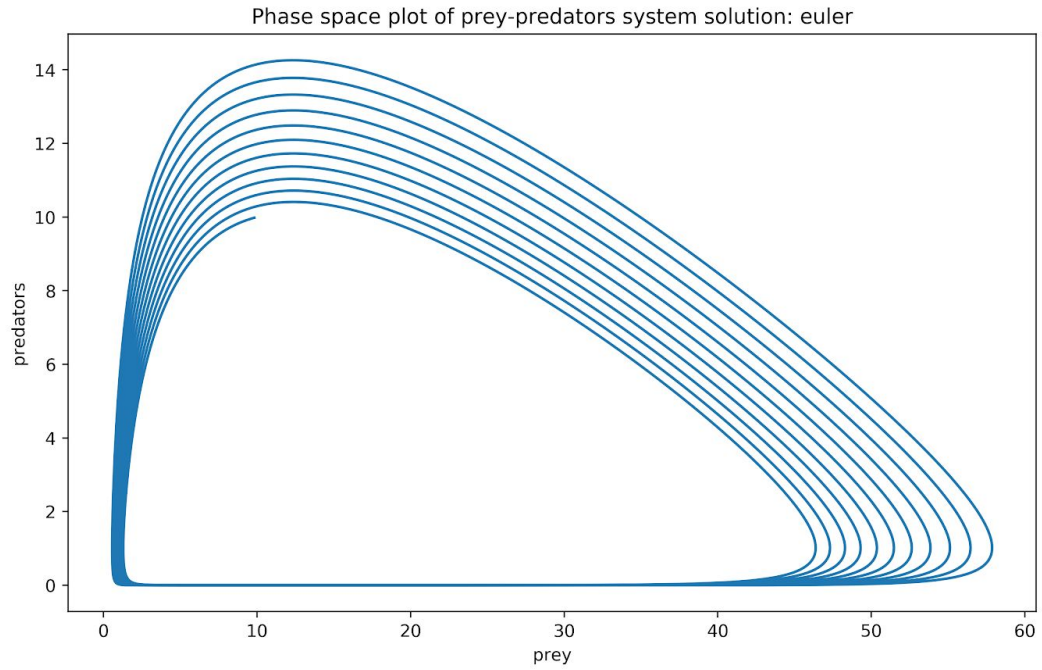


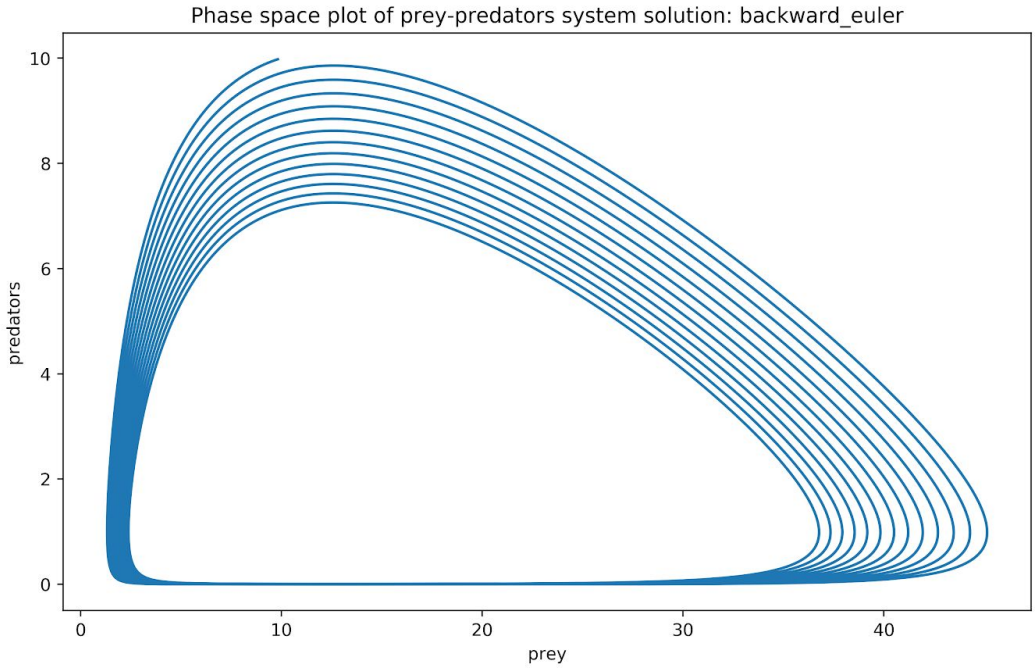
Metody dające rozwiązania nie do końca zgodne z oczekiwaniami: Euler, Backward Euler

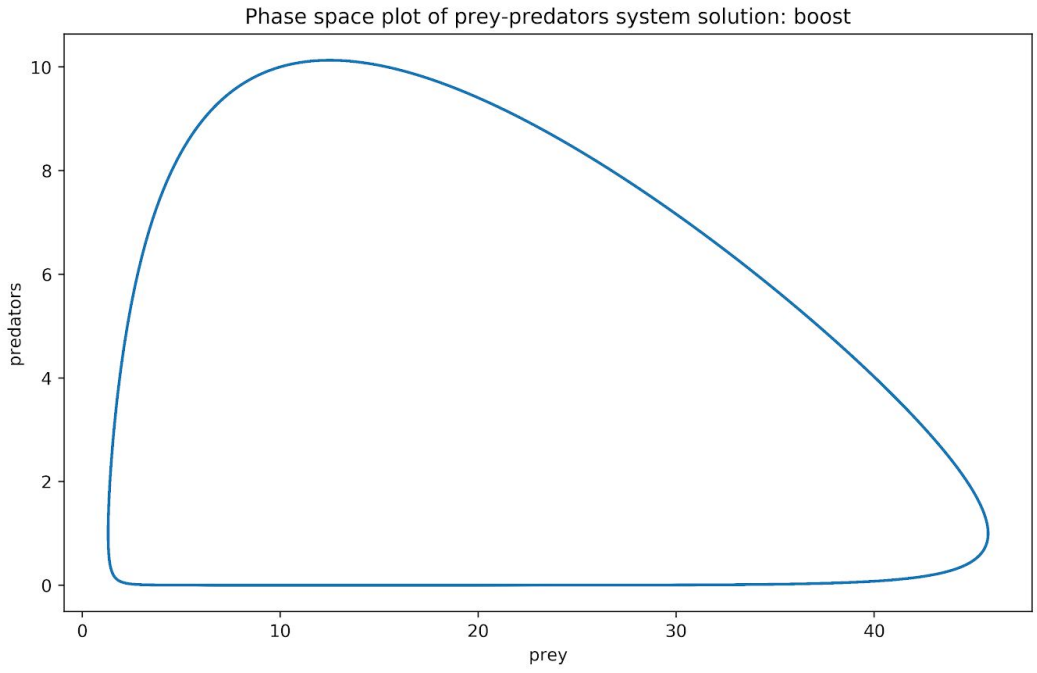




Choć powyższe wykresy na oko nie różnią się znacząco od rozwiązań przy pomocy RK2, RK4 czy Boosta, to różnicę łatwo zauważyć na wykresie przestrzeni fazowej.







Wnioski:
Jak łatwo zauważyć metoda Eulera (w obydwu wariantach) nie prowadzi do zachowania energii w układzie. Pozostałe metody prowadzą do wyników zgodnych z oczekiwanymi.

c. (zad5) Rozpad promieniotwórczy

Rozpad promieniotwórczy to samorzutna przemiana, w wyniku, której jądro promieniotwórcze zamienia się w inne jądro (które również może być promieniotwórcze). Procesowi temu towarzyszy wyemitowanie promieniowania.

Rozpad promieniotwórczy opisuje poniższe równanie:

$$\frac{\partial u}{\partial t} + \frac{u}{\tau} = 0$$

dla $u_0 = 1$. Analityczne rozwiązanie może być uzyskane z następującego wzoru:

$$u = e^{-\frac{t}{\tau}} .$$

Kod:

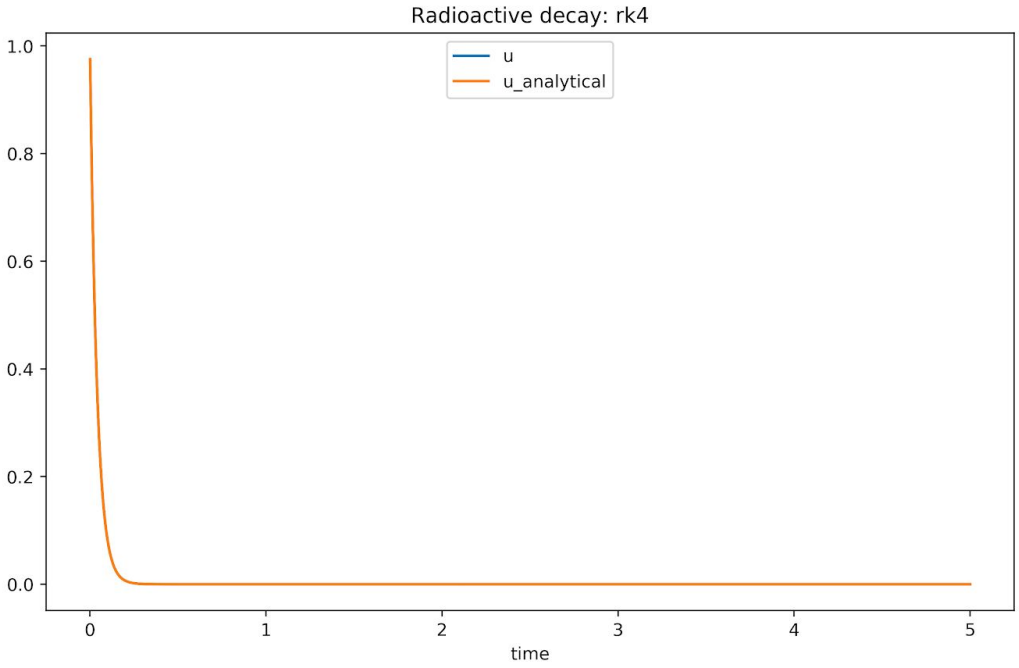
```
class DecayODE : public IDifferentialEquation
{
    public:
    DecayODE(std::vector<double> constants) :
        IDifferentialEquation(constants) {}

        std::vector<double> getDerivatives(std::vector<double> variables)
        {
            double u = variables[0];

            std::vector<double> derivatives({
                (-1) * u / getTau()
            });

            return derivatives;
        }
        int getDimensions()
        {
            return 1;
        }

        private:
        double getTau()
        {
            return constants[0];
        }
};
```

Wyniki:

Przypadek 1

$\tau = \frac{1}{25}$, $\Delta t = 0.001$, $\Delta t \leq 2\tau$

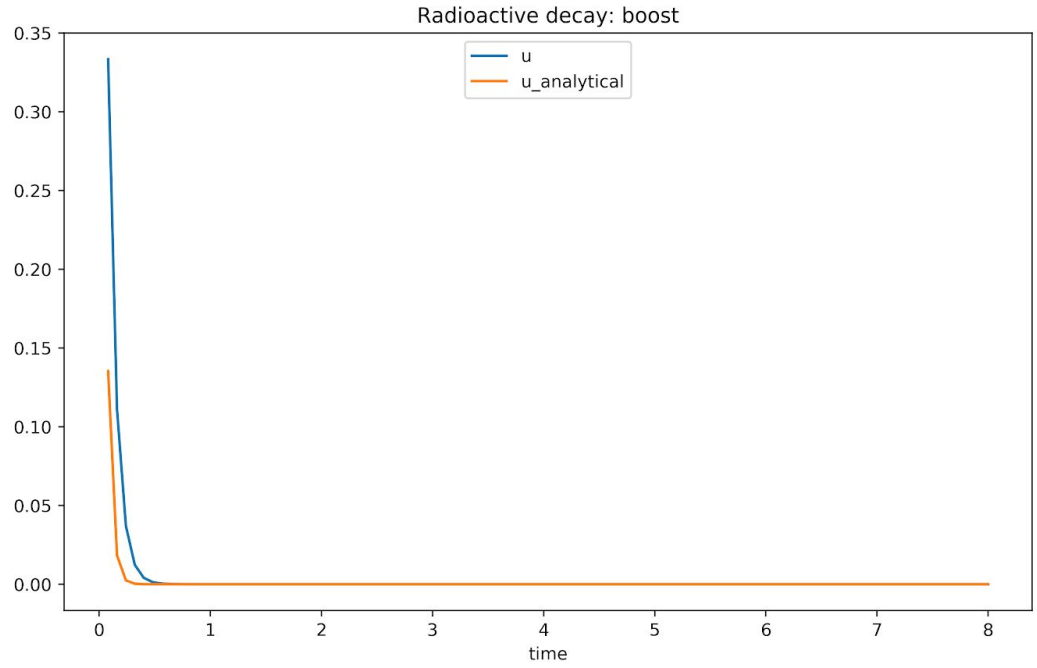
Metody dające rozwiązania zgodne z oczekiwaniami: wszystkie



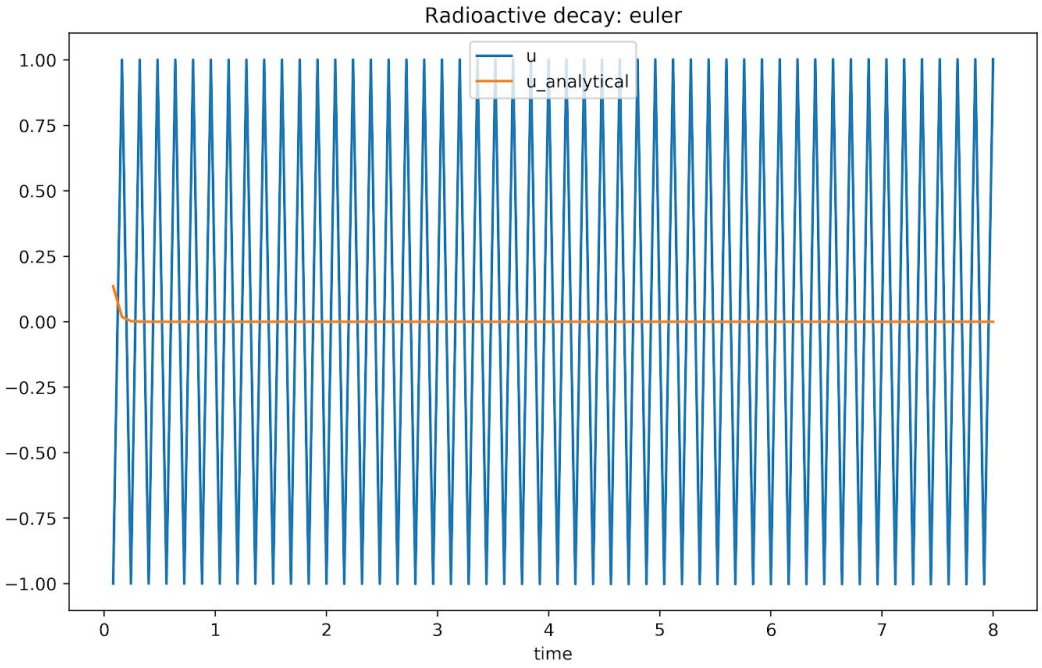
Przypadek 2

$\tau = \frac{1}{25}$, $\Delta t = 0.09$, $\Delta t > 2\tau$

Metody dające poprawne rozwiązania: Boost



Metody dające rozwiązania zgodne z oczekiwaniami (lecz błędne): pozostałe



Wnioski:
Gdy wykorzystany jest krok czasowy gwarantujący stabilność ($\Delta t \leq 2\tau$) wszystkie metody prowadzą do wyników zgodnych z uzyskanymi analitycznie. W przeciwnym wypadku udało się to jedynie przy pomocy biblioteki Boost.

d. (zad6) Drganie sprężyny

Ruch harmoniczny prosty opisuje równanie:

$$m \cdot \frac{\partial x^2}{\partial t^2} = - k \cdot x \,,$$

gdzie m to masa ciała, a k to stała sprężystości.

Kod:

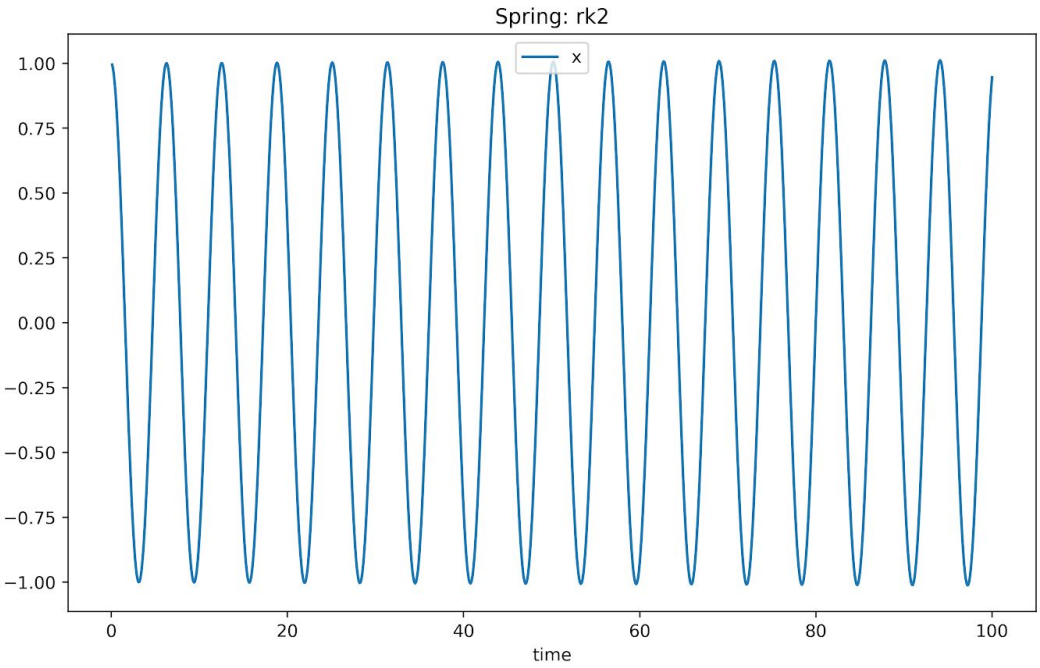
```
class SpringODE : public IDifferentialEquation
{
    public:
    SpringODE(std::vector<double> constants) :
        IDifferentialEquation(constants) {}
```

```
    std::vector<double> getDerivatives(std::vector<double> variables)
    {
        double y1 = variables[0];
        double y2 = variables[1];

        std::vector<double> derivatives({
            y2,
            (-1) * y1 * getK() / getM()
        });
        return derivatives;
    }

    int getDimensions()
    {
        return 2;
    }

    private:
    double getK()
    {
        return constants[0];
    }
    double getM()
    {
        return constants[1];
    }
};
```

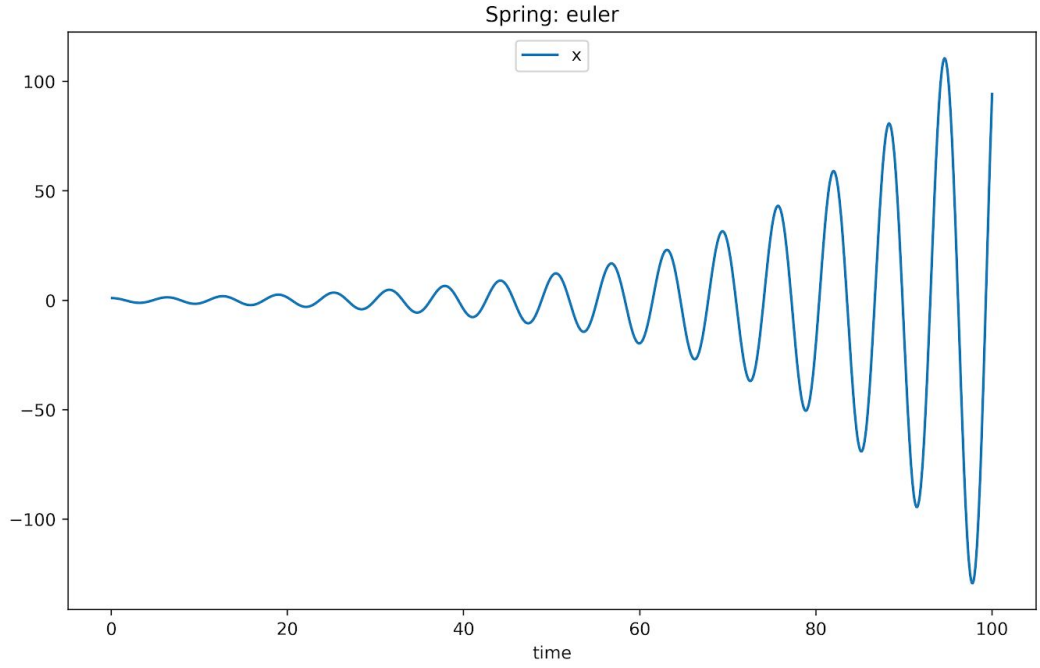
Wyniki:

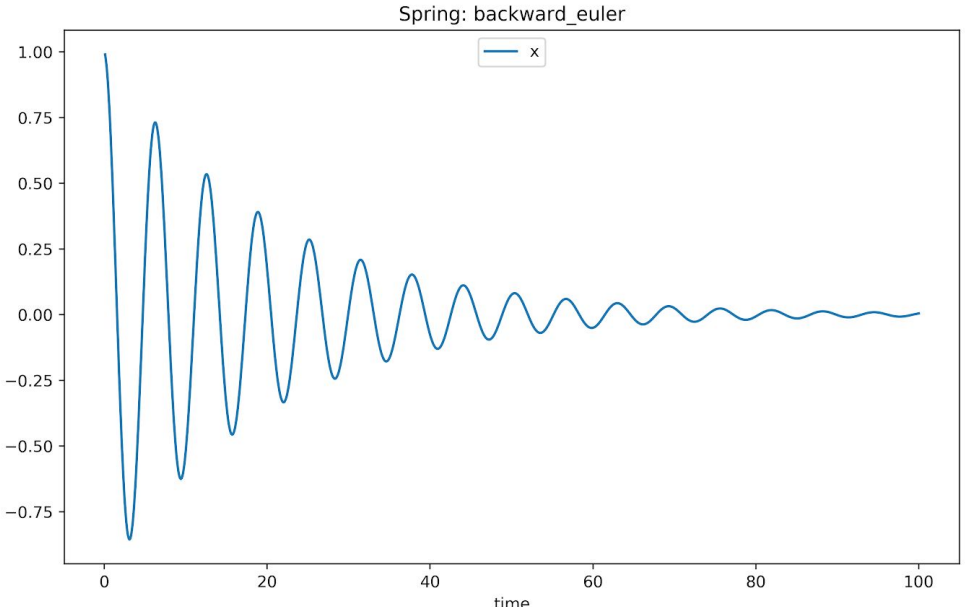
$k = 1,\ m = 1$

Metody dające poprawne rozwiązania: RK2, RK4, Boost



Metody dające niepoprawne rozwiązania: Euler, Backward Euler





Wnioski:

Podobnie jak w powyższych przypadkach metody Eulera nie doprowadziły do poprawnego rozwiązania. Biorą one pod uwagę zbyt mały fragment historii układu i nie pozwalają na zachowanie energii.