Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Modele systemów dynamicznych

Sprawozdanie z laboratorium 3

Igor Lis

Nr albumu: **284053**

Kierunek: **Inżynieria systemów**

WROCŁAW 2025

1. Wstęp teoretyczny

Symulujemy znane z poprzednich list układy Lotki-Volterry i Lorenza na dwa sposoby:

- przy użyciu metody Eulera,
- przy użyciu metody integrate.odeint z pakietu scipy.

Układ Lotki-Volterry definiujemy jako

$$\begin{cases} \frac{\mathrm{d}x}{\mathrm{d}t} = (a - by)x \\ \frac{\mathrm{d}y}{\mathrm{d}t} = (cx - d)y, \end{cases}$$

gdzie *x* - populacja ofiar, *y* - populacja drapieżników, *t* - czas, *a* - częstość narodzin ofiar, *b* - częs tość umierania ofiar, *c* - częstość narodzin drapieżników, *d* - częstość umierania drapieżników. Możemy przyjąć następujące wartości parametrów: $a = 1.2$, $b = 0.6$, $c = 0.3$, $d = 0.8$ oraz populacje początkowe $x0 = 2$ i $y0 = 1$.

Układ Lorenza zdefiniowany jest za pomocą układu trzech równań różniczkowych:

$$\begin{cases} \frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x) \\ \frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y \\ \frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z, \end{cases}$$

Gdzie $\sigma = 10$, $\beta = 3/8$ i $\rho = 28$ oraz warunki początkowe $x(0) = y(0) = z(0) = 1$.

2. Opis rozwiązania

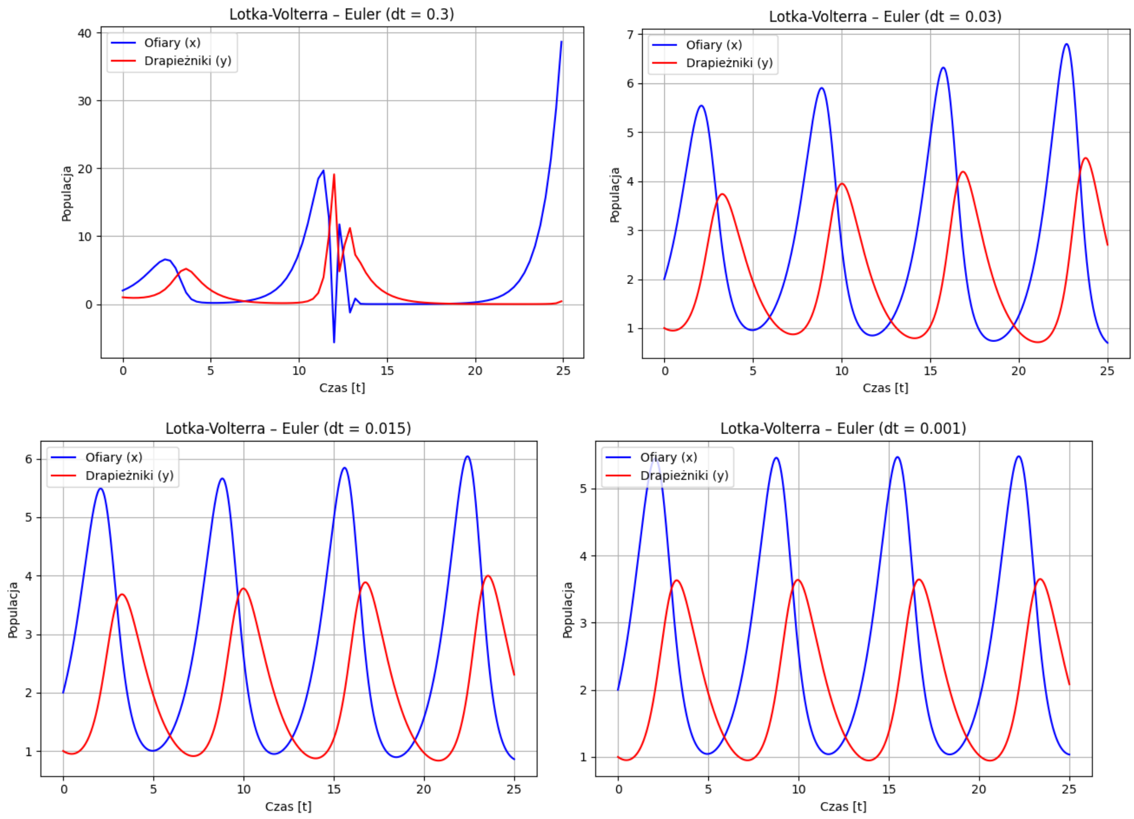
```
# Układ Lotki-Volterry:
def lotka_volterra(X, t, a=1.2, b=0.6, c=0.3, d=0.8):
    x, y = X
    dxdt = (a - b * y) * x
    dydt = (c * x - d) * y
    return [dxdt, dydt]


# Układ Lorenza:
def lorenz(X, t, sigma=10, beta=8 / 3, rho=28):
    x, y, z = X
    dxdt = sigma * (y - x)
    dydt = x * (rho - z) - y
    dzdt = x * y - beta * z
    return [dxdt, dydt, dzdt]
```

```
# ==== Własna implementacja metody Eulera ====
def euler(f, X0, t):
    if len(t) < 2:
         return np.array([X0]) if len(t) > 0 else np.array([])

    dt = t[1] - t[0]  # krok czasowy (zakładamy stały krok)
    X = np.zeros((len(t), len(X0)))  # macierz na wynik
    X[0] = np.array(X0)  # warunki początkowe

    for i in range(1, len(t)):
        dX = np.array(f(X[i - 1], t[i - 1]))

        if np.any(np.isnan(dX)) or np.any(np.isinf(dX)) or np.any(np.abs(dX) >
1e10):
             print(f"Euler warning/error in step {i} for dt={dt}: dX = {dX}.
Truncating result.")
             return X[:i]

        X[i] = X[i - 1] + dt * dX

        if (
            np.any(np.isnan(X[i]))
            or np.any(np.isinf(X[i]))
            or np.any(np.abs(X[i]) > 1e10)
        ):
            print(f"Euler warning/error in step {i} for dt={dt}: X = {X[i]}.
Truncating result.")
            return X[:i]

    return X
```

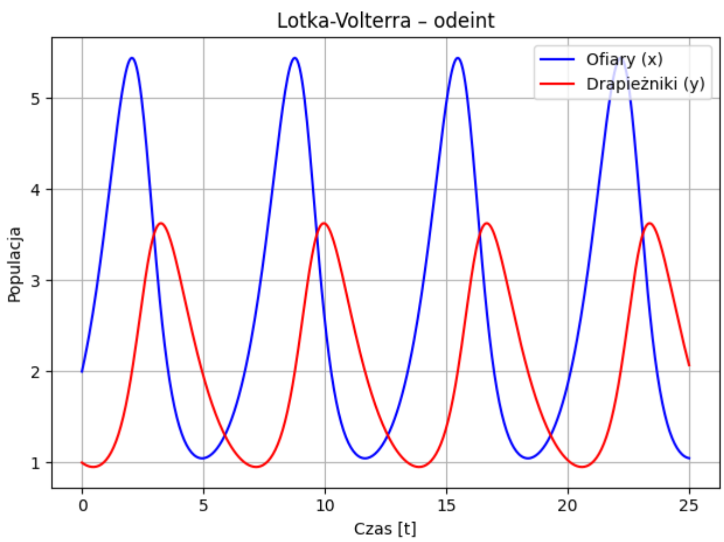
```python
def calculate_euler_error(system, initial_conditions, t_values):
    euler_solution = euler(system, initial_conditions, t_values)

    if len(euler_solution) == 0:
         print("Euler simulation returned no points. Cannot calculate error.")
         return np.nan, np.array([]), np.array([])

    t_truncated = t_values[: len(euler_solution)]
    if len(t_truncated) == 0:
        print("Truncated time vector is empty. Cannot calculate error.")
        return np.nan, np.array([]), np.array([])

    odeint_solution = odeint(system, initial_conditions, t_truncated, rtol=1e-8,
atol=1e-8)

    if odeint_solution.shape[0] != len(t_truncated):
         print(f"Warning: odeint returned {odeint_solution.shape[0]} points,
expected {len(t_truncated)}. Skipping error calculation.")
         return np.nan, np.array([]), np.array([])

    errors = np.sqrt(
        np.sum((euler_solution - odeint_solution) ** 2, axis=1)
    )

    mean_error = np.mean(errors) if len(errors) > 0 else np.nan

    return mean_error, errors, t_truncated
```

3. Wyniki obliczeń

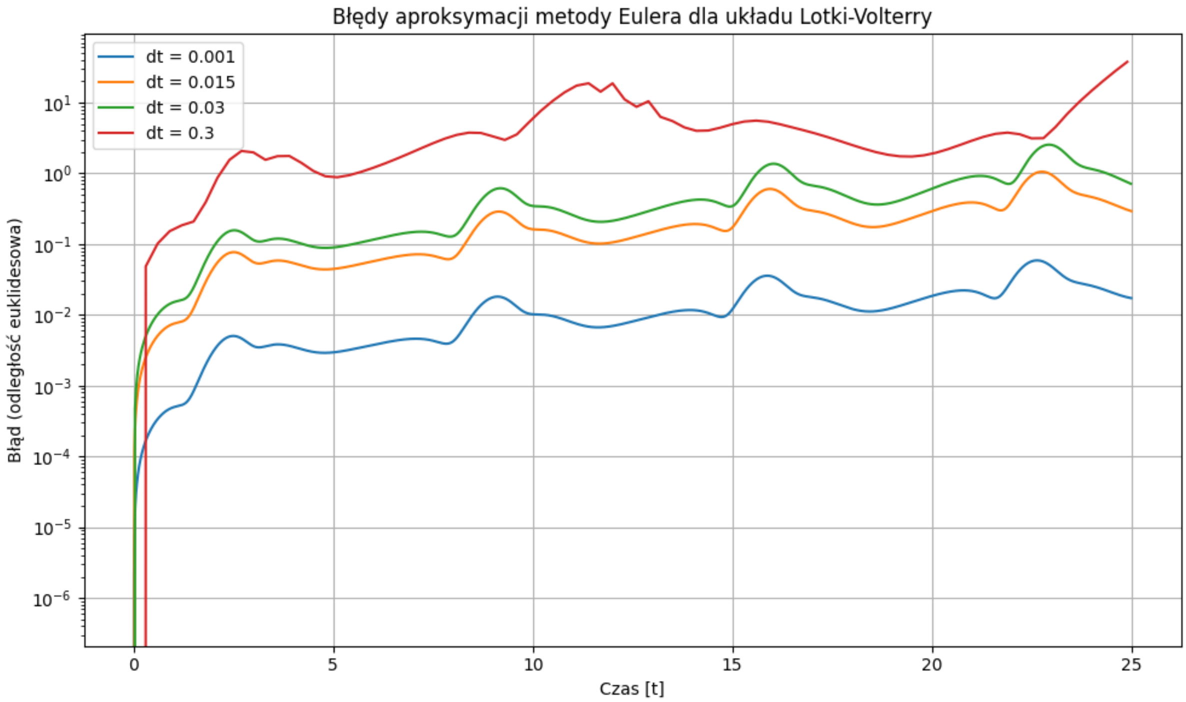
Model Lotka-Volterra



Rysunek 1: Model Lotki-Volterry przy użyciu metody Eulera dla czterech różnych kroków symulacji (0.3, 0.03, 0.015, 0.001).

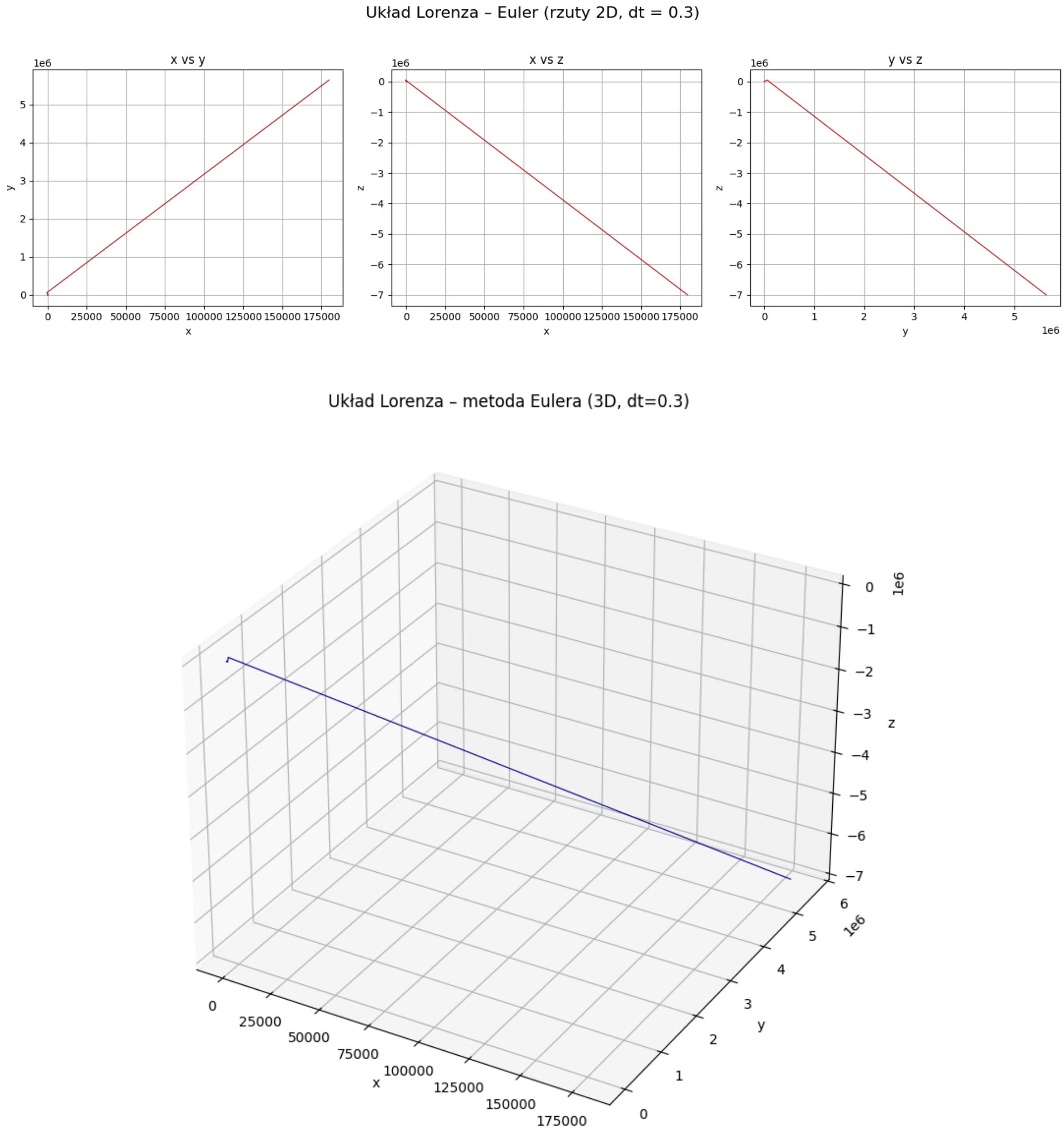


Rysunek 2: Model Lotki-Volterry przy użyciu metody odeint.

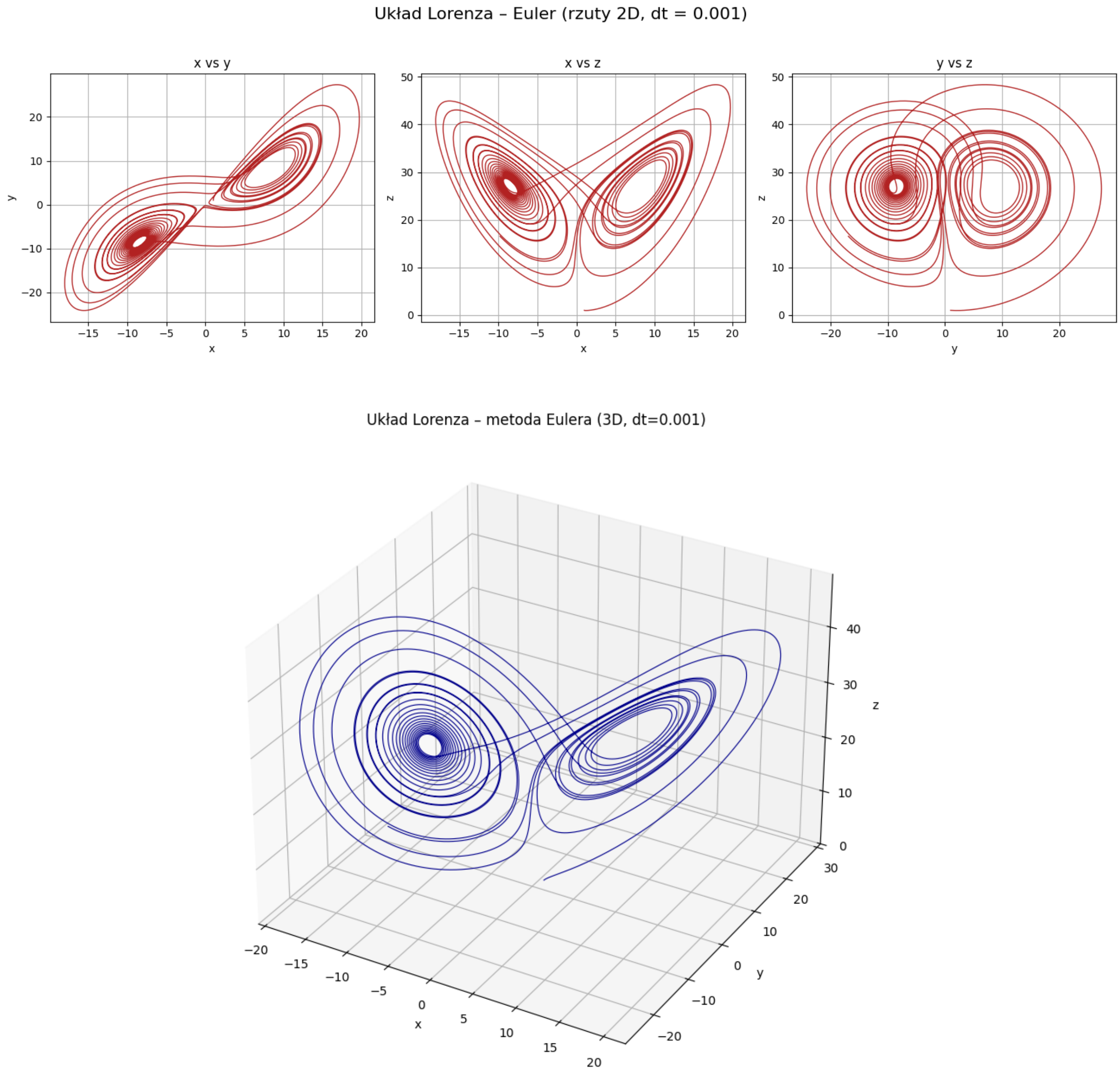


Rysunek 3: Błędy aproksymacji układu Lotki-Volterry dla metody Eulera

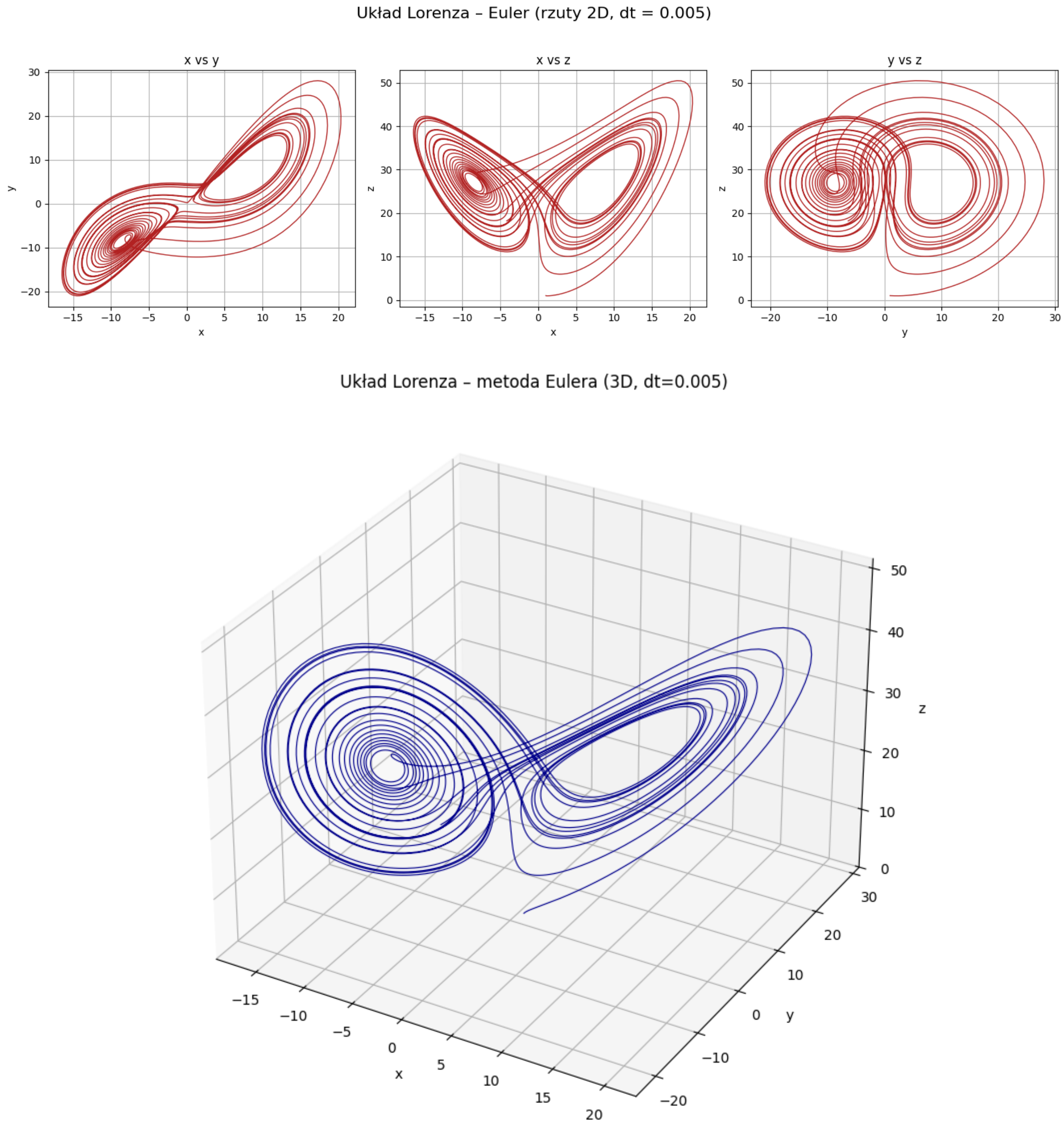
Układ Lorenza



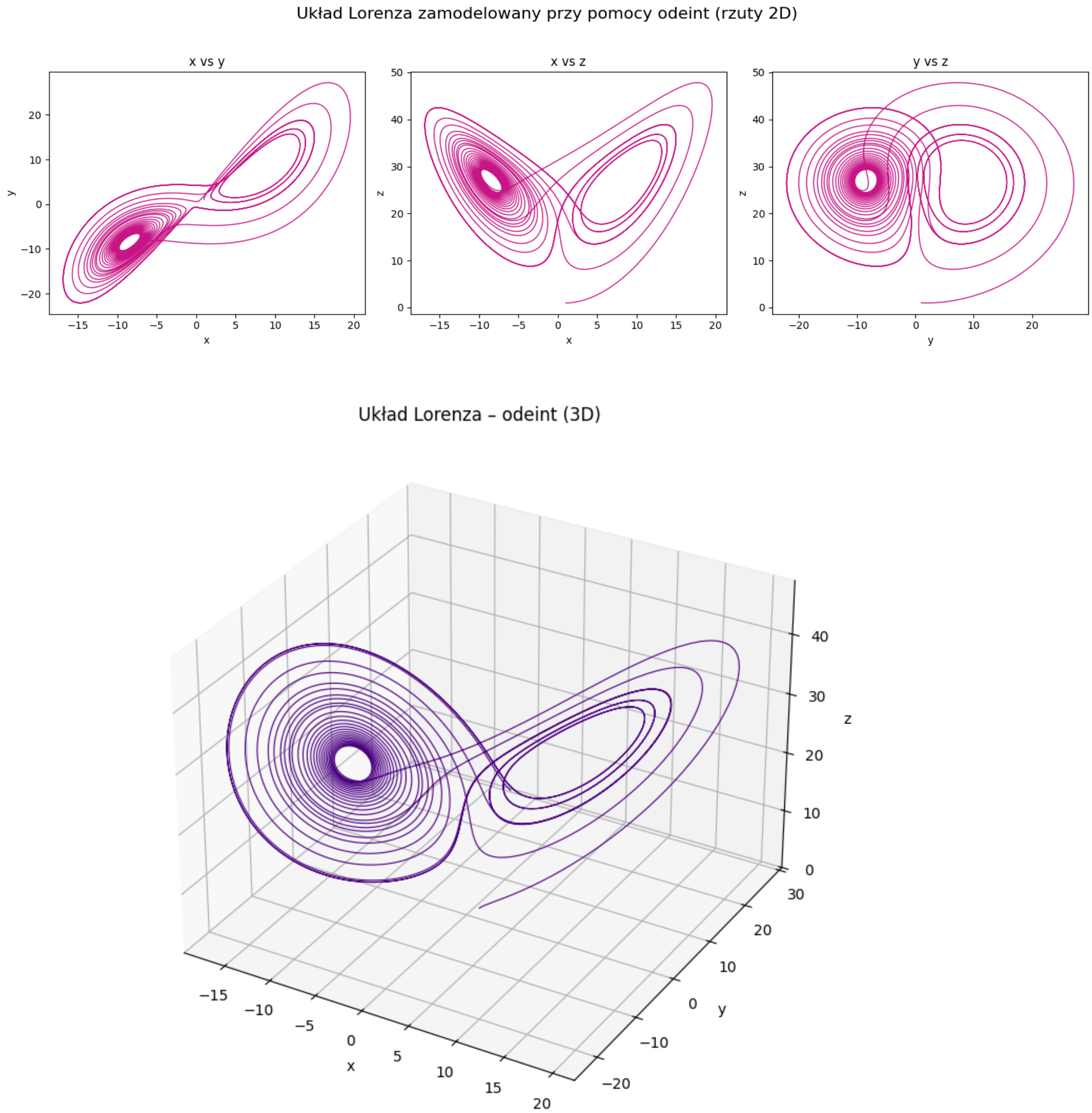
Rysunek 4: Układ Lorenza przy użyciu metody Eulera dla kroku symulacji dt = 0.3.



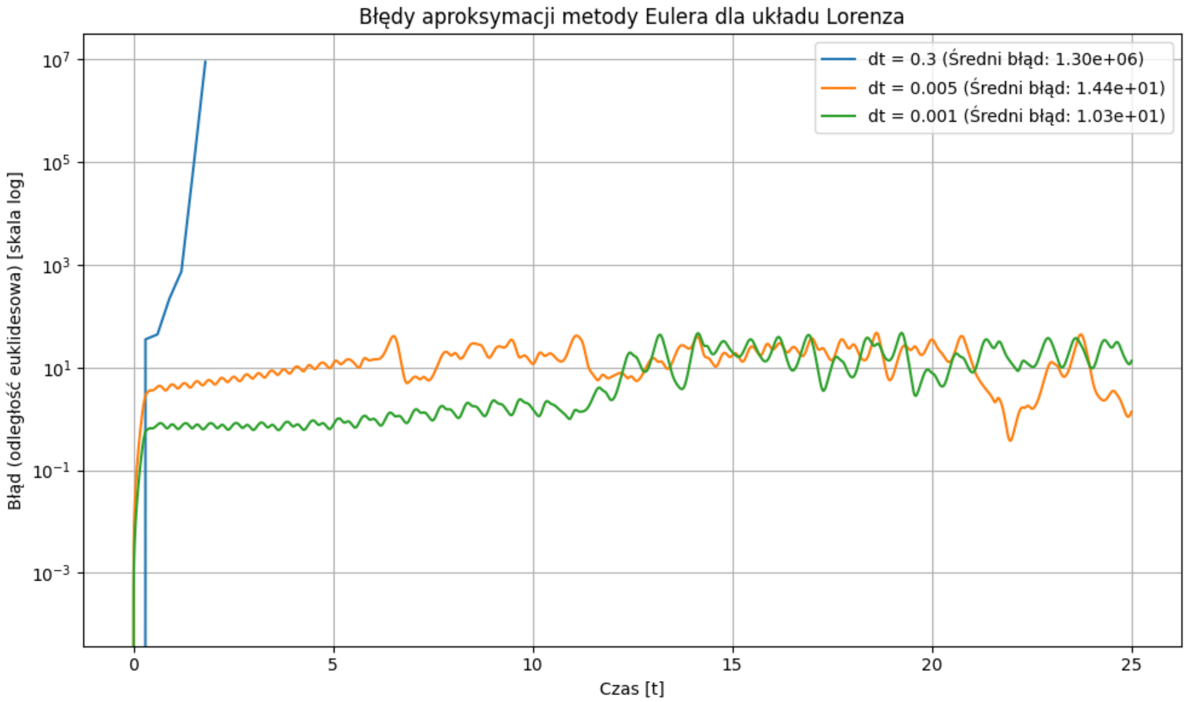
Rysunek 5: Układ Lorenza przy użyciu metody Eulera dla kroku symulacji dt = 0.001.



Rysunek 6: Układ Lorenza przy użyciu metody Eulera dla kroku symulacji dt = 0.005.



Rysunek 7: Układ Lorenza w 3D przy użyciu odeint.



Rysunek 8: Błędy aproksymacji układu Lorenza dla metody Eulera

4. Wnioski i podsumowanie

- **Metody Numeryczne**: Metoda Eulera jest prostą metodą o stałym kroku czasowym (dt), łatwą do zaimplementowania, ale charakteryzującą się niską dokładnością. odeint używa znacznie bardziej zaawansowanych, adaptacyjnych algorytmów, które automatycznie dobierają krok czasowy, co zapewnia wyższą dokładność i stabilność.

- **Wpływ dt (dla Eulera)**: Dokładność rozwiązania uzyskanego metodą Eulera silnie zależy od wielkości kroku czasowego dt. Zmniejszenie dt zwiększa dokładność i opóźnia wystąpienie niestabilności numerycznej, co jest wyraźnie widoczne na wykresach błędów dla obu układów.

- **Kumulacja Błędów**: Błędy numeryczne wprowadzane w każdym kroku metody Eulera kumulują się w czasie, powodując stopniowe odchylenie od dokładnego rozwiązania. Wykresy błędów pokazują, jak błąd narasta w miarę trwania symulacji.

- **Układ Lotki-Volterry**: Dla stabilnego, okresowego układu Lotki-Volterry, metoda Eulera może jakościowo odtworzyć dynamikę populacji przy odpowiednio małym dt, ale wykazuje dryf (np. zmiany amplitudy). odeint zapewnia stabilne i wierne rozwiązanie bez tego dryfu. Duże dt powoduje niestabilność Eulera dla LV.

- **Układ Lorenza (Chaotyczny)**: Układ Lorenza jest znacznie trudniejszy do numerycznego modelowania ze względu na jego chaotyczny charakter i wrażliwość na błędy.

- **Duże dt** (np. 0.3): Metoda Eulera całkowicie zawodzi dla dużego dt. Błędy są tak duże, że trajektoria natychmiast dywaguje od atraktora Lorenza, "uciekając" w linii lub powodując przerwanie symulacji, co tłumaczy brak "efektu motyla" na rysunku 4.

- **Mniejsze d**t: Mniejsze dt pozwalają Eulerowi na przybliżenie struktury atraktora Lorenza przez pewien czas, ale błędy numeryczne nadal narastają (wykładniczo, co widać na wykresie błędu), ostatecznie powodując, że obliczona trajektoria odbiega od rzeczywistej trajektorii (mimo że może pozostać w obszarze atraktora).

- **Odeint dla układu Lorenza**: odeint skutecznie radzi sobie z chaotyczną dynamiką Lorenza, wiernie odwzorowując atraktor dzięki automatycznemu dostosowywaniu kroku w celu kontroli błędu.

- **Wniosek końcowy**: Proste metody o stałym kroku, jak Euler, są niewystarczające i numerycznie niestabilne dla złożonych, a zwłaszcza chaotycznych układów dynamicznych, chyba że używane są ekstremalnie małe kroki czasowe (co jest nieefektywne). Do wiarygodnego modelowania takich układów konieczne jest stosowanie bardziej zaawansowanych, adaptacyjnych metod numerycznych (jak te w odeint).