Laboratorium 12
Metody Numeryczne

Instrukcja:
Na zajęciach należy wykonać poniższe zadania, a następnie sporządzić sprawozdanie zawierające odpowiedzi z komentarzami.

Cel zajęć: Celem zajęć jest zapoznanie się z numerycznymi metodami rozwiązywania równań różniczkowych zwyczajnych. Będziemy rozpatrywać równania różniczkowe postaci

$$\dot{x}(t) = f(x(t), t)$$

gdzie:

$x(t) \in \mathbb{R}^n$,

$t \geq 0$

z warunkiem początkowym $x(0) = x0$

Jest to tak zwany problem początkowy (problem Cauchy'ego) dla równań różniczkowych zwyczajnych.

Poniej zaimplementuj funkcje niezbędne do wykonania laboratoriów:

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
from typing import Union, Callable

def solve_euler(fun: Callable, t_span: np.array, y0: np.array):
    '''
    Funkcja umożliwiająca rozwiązanie układu równań różniczkowych z wykorzystaniem met
    
    Parameters:
    fun: Prawa strona równania. Podana funkcja musi mieć postać fun(t, y).
    Tutaj t jest skalarem i istnieją dwie opcje dla ndarray y: Może mieć kształt (n,)
    Alternatywnie może mieć kształt (n, k); wtedy fun musi zwrócić tablicę typu array_
    t_span: wektor czasu dla którego ma zostać rozwiązane równanie
    y0: warunek początkowy równania o wymiarze (n,)
    
    Returns:
    np.array: macierz o wymiarze (n, m) zawierająca w kolumnach kolejne rozwiązania fu
    '''
    return None

def arenstorf(t, x: np.array):
    '''
    Parameters:
    t: czas
    x: wektor stanu
    Results:
    (np.array): wektor pochodnych stanu
    '''
    return None
```

Zadanie 1.
Zaimplementuj metodę *solve_euler* z main.py

In [2]:
```
# Definicja równania różniczkowego dy/dx = x + y + xy
def func(x, y):
    return (x + y + x * y)

# Implementacja metody Eulera do przybliżania rozwiązania
def solve_euler(x0, y, h, x):
    temp = -0 #zmienna tymczasowa, bo niekonieczne jest wpisywanie y

    # Pętla wykonuje się do momentu osiągnięcia punktu, w którym chcemy przybliżyć roz
    while x0 < x:
        temp = y
        y = y + h * func(x0, y)
        x0 = x0 + h

    # Wydruk przybliżonego rozwiązania
    print("Przybliżone rozwiązanie przy x =", x, " to ", "%.6f" % y)
```

Zadanie 2.

Dla 3 różnych kroków czasowych (1e1, 1e-2, 1e-5) korzystając z metody z zadania 1 rozwiąż równanie

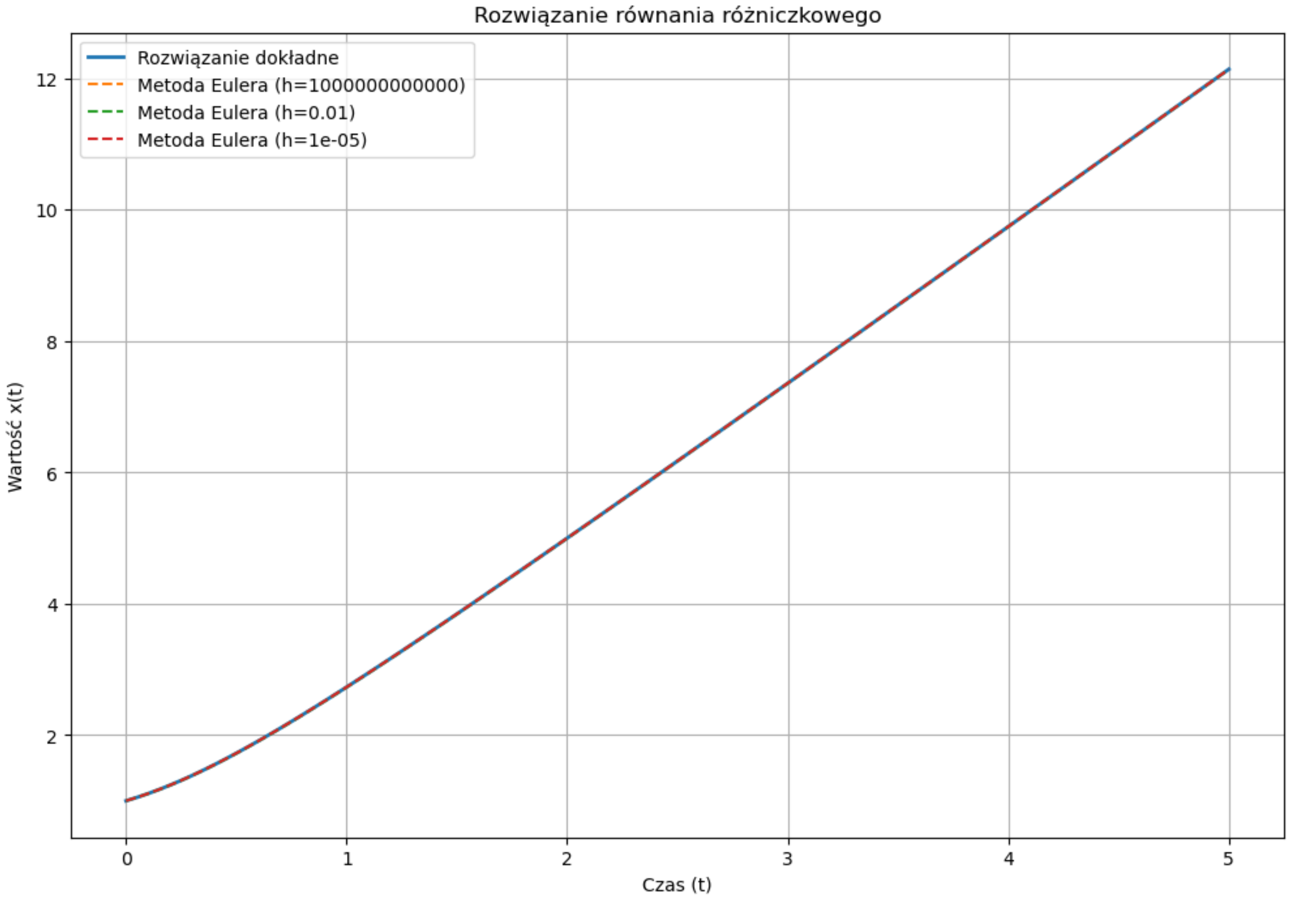
$\dot{x}(t) = \frac{x+t}{x-t}$

$x(0) = 1$ (równanie to posiada rozwiązanie dokładne: $x(t) = t + \sqrt{1 + 2t^2}$).

Narysuj wykres podanego rozwiązania dokładnego oraz uzyskanych rozwiązań numerycznych.

In [14]:

```python
# Definicja równania różniczkowego
def func(x, t):
    return (x + t) / (x - t)

# Rozwiązanie dokładne
def exact_solution(t):
    return t + np.sqrt(1 + 2 * t**2)

# Implementacja metody Eulera do przybliżania rozwiązania
def solve_euler(x0, t, h):
    result = []
    for ti in t:
        result.append(x0)
        x0 = x0 + h * func(x0, ti)
    return np.array(result)

# Warunki początkowe
x0 = 1
t_exact = np.linspace(0, 5, 100)  # Zakres czasowy dla rozwiązania dokładnego

# Różne kroki czasowe
h_values = [1e1, 1e-2, 1e-5]

# Rysowanie wykresów
plt.figure(figsize=(12, 8))

# Wykres rozwiązania dokładnego
plt.plot(t_exact, exact_solution(t_exact), label="Rozwiązanie dokładne", linewidth=2)

# Wykresy rozwiązań numerycznych dla różnych kroków czasowych
for h in h_values:
    t_numeric = np.arange(0, 5, h)
    x_numeric = solve_euler(x0, t_numeric, h)
    plt.plot(t_numeric, x_numeric, label=f"Metoda Eulera (h={h})", linestyle='dashed'

# Konfiguracja wykresu
plt.title("Rozwiązanie równania różniczkowego")
plt.xlabel("Czas (t)")
plt.ylabel("Wartość x(t)")
plt.legend()
plt.grid(True)
plt.show()
```



Zadanie 3.

Dla 3 różnych kroków czasowych (1e1, 1e-2, 1e-5):

1. Rozwiąż układ równań różniczkowych:

$$\dot{x}_1(t) = x_3(t)$$
$$\dot{x}_2(t) = x_4(t)$$
$$\dot{x}_3(t) = -\frac{x_1(t)}{(x_1(t)^2+x_2(t)^2)^{\frac{3}{2}}}$$
$$\dot{x}_4(t) = -\frac{x_2(t)}{(x_1(t)^2+x_2(t)^2)^{\frac{3}{2}}}$$

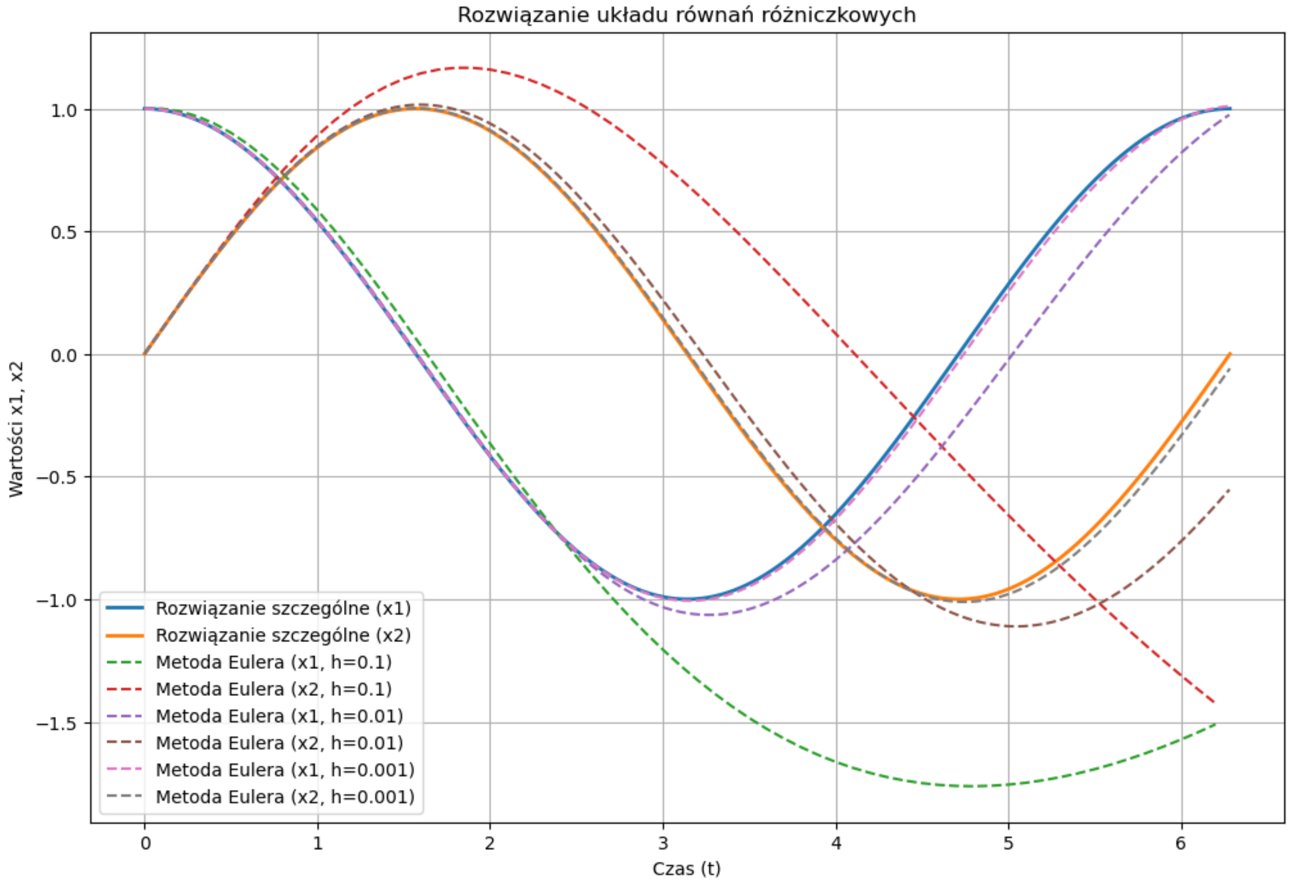
z warunkiem początkowym $x(0) = [1, 0, 0, 1]^T$.

Dla takiego warunku początkowego układ ten ma rozwiązanie szczególne

$$x(t) = \begin{bmatrix} cos(t) \\ sin(t) \\ -sin(t) \\ cos(t) \end{bmatrix}.$$

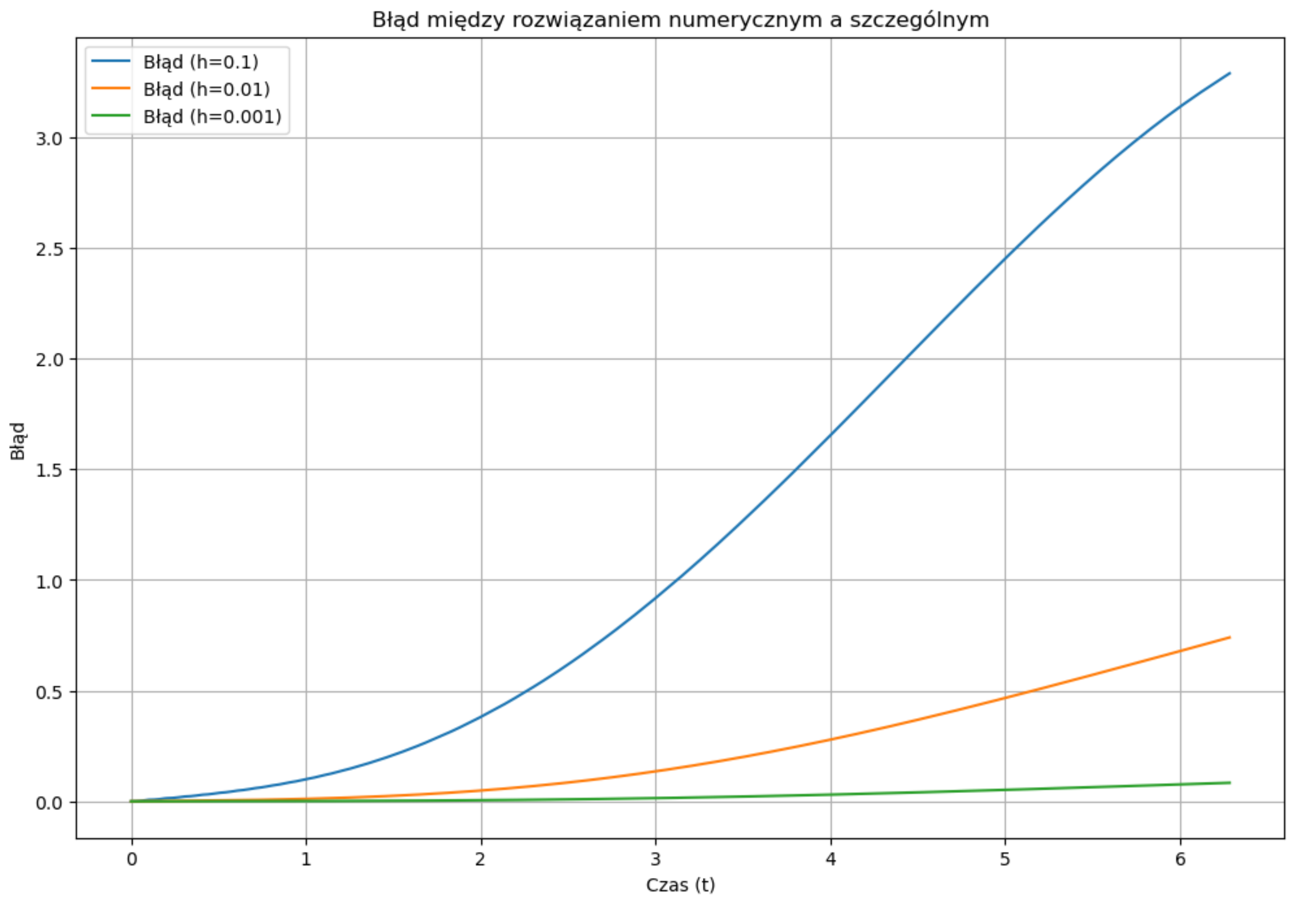
2. Narysuj wykres podanego rozwiązania szczegółowego oraz uzyskanych rozwiązań numerycznych.
3. Sprawdź po jakim czasie t rozwiązania numeryczne przestaną dawać zadowalające wyniki. W tym celu zbadaj błąd między rozwiązania numerycznego względem szczagólnego

In [5]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Definicja układu równań różniczkowych
def system_equations(x, t):
    x1_dot = x[2]
    x2_dot = x[3]
    x3_dot = -x[0] / (x[0]**2 + x[1]**2)**(3/2)
    x4_dot = -x[1] / (x[0]**2 + x[1]**2)**(3/2)
    return np.array([x1_dot, x2_dot, x3_dot, x4_dot])

# Implementacja metody Eulera do rozwiązania układu równań różniczkowych
def solve_euler_system(initial_conditions, t, h):
    result = [initial_conditions]
    x = np.array(initial_conditions)

    for ti in t[1:]:
        x = x + h * system_equations(x, ti)
        result.append(x)

    return np.array(result)

# Warunki początkowe
initial_conditions = [1, 0, 0, 1]

# Zakres czasowy
t_exact = np.linspace(0, 2*np.pi, 1000)  # Rozwiązanie szczególne

# Różne kroki czasowe
h_values = [1e-1, 1e-2, 1e-3]

# Rysowanie wykresów
plt.figure(figsize=(12, 8))

# Wykres rozwiązania szczególnego
x_exact = np.array([[np.cos(ti), np.sin(ti), -np.sin(ti), np.cos(ti)] for ti in t_exa
plt.plot(t_exact, x_exact[:, 0], label="Rozwiązanie szczególne (x1)", linewidth=2)
plt.plot(t_exact, x_exact[:, 1], label="Rozwiązanie szczególne (x2)", linewidth=2)

# Wykresy rozwiązań numerycznych dla różnych kroków czasowych
for h in h_values:
    t_numeric = np.arange(0, 2*np.pi, h)
    x_numeric = solve_euler_system(initial_conditions, t_numeric, h)
    plt.plot(t_numeric, x_numeric[:, 0], linestyle='dashed', label=f"Metoda Eulera (x
    plt.plot(t_numeric, x_numeric[:, 1], linestyle='dashed', label=f"Metoda Eulera (x

# Konfiguracja wykresu
plt.title("Rozwiązanie układu równań różniczkowych")
plt.xlabel("Czas (t)")
plt.ylabel("Wartości x1, x2")
plt.legend()
plt.grid(True)
plt.show()
```



Zadanie 3.3 Obliczamy błąd między rozwiązaniem numerycznym, a rozwiązaniem szczególnym w zależności od czasu dla różnych kroków czasowych.

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# ... (kod wcześniejszy)

# Obliczenie błędu w zależności od czasu
errors = []

for h in h_values:
    t_numeric = np.arange(0, 2*np.pi, h)
    x_numeric = solve_euler_system(initial_conditions, t_numeric, h)

    # Interpolacja rozwiązania numerycznego na czas rozwiązania szczególnego
    interp_func = interp1d(t_numeric, x_numeric.T, kind='linear', fill_value="extrapol
    x_numeric_interp = interp_func(t_exact).T

    # Obliczenie błędu dla każdej chwili czasu
    error = np.linalg.norm(x_exact - x_numeric_interp, axis=1)
    errors.append(error)

# Rysowanie wykresu błędu
plt.figure(figsize=(12, 8))

for i, h in enumerate(h_values):
    plt.plot(t_exact, errors[i], label=f"Błąd (h={h})")

plt.title("Błąd między rozwiązaniem numerycznym a szczególnym")
plt.xlabel("Czas (t)")
plt.ylabel("Błąd")
plt.legend()
plt.grid(True)
plt.show()
```



Wnioski: Mozemy zauwazyc coraz mniejszy błąd z zwiekszaniem kroku.

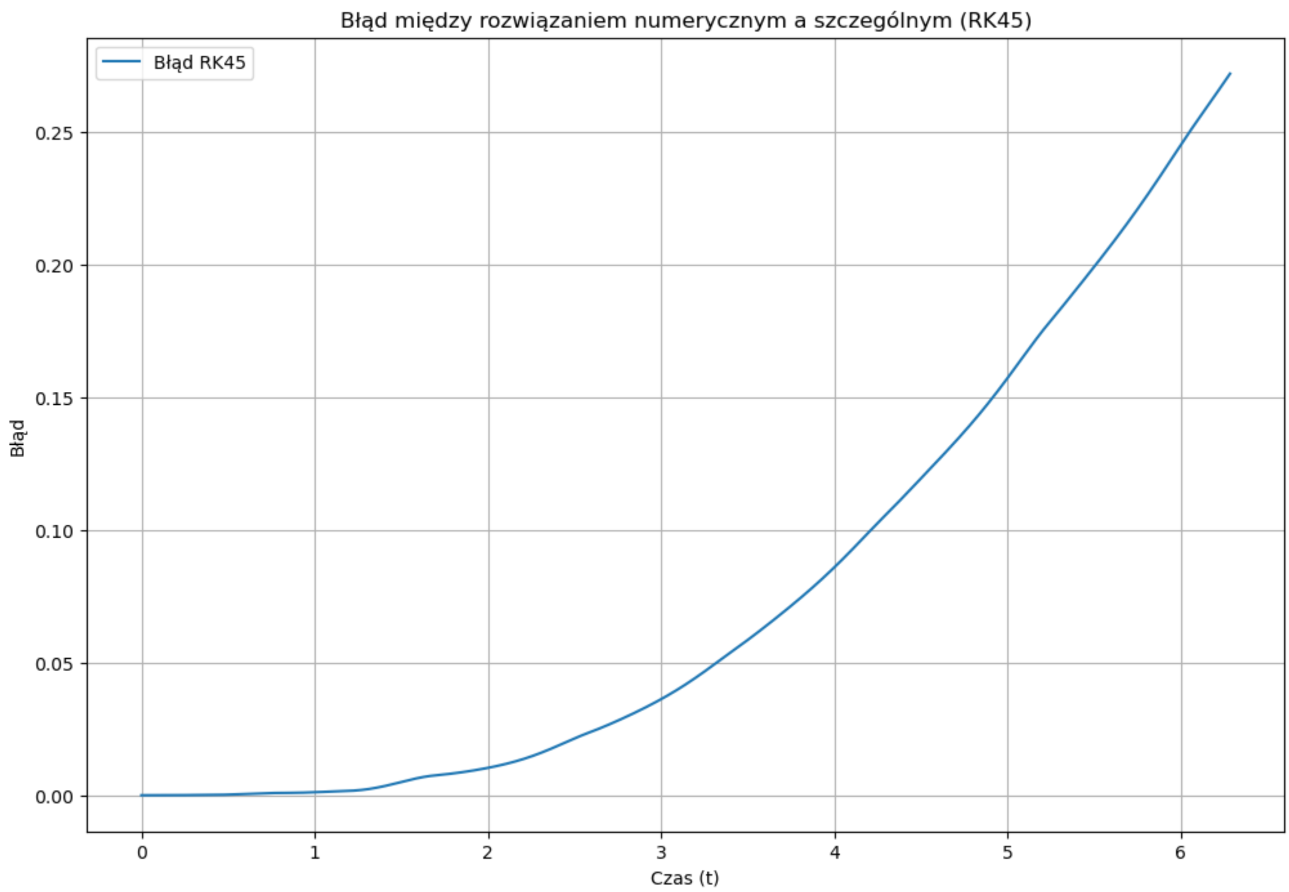
Zadanie 4.

Za pomocą funkcji solve_ivp przy wykorzystaniu dwóch metod RK45 i RK23 rozwiąż układ równań z poprzedniego zadania i porównaj wyniki dla takiego samego przedziału czasu.

Sprawdź ile razy każda z metod obliczała równanie prawej strony (parametr nfev). Odnieś to do analogicznej liczby z metody z poprzedniego zadania

Dla wybranej metody zbadaj wpływ parametru rtol na rozwiązanie

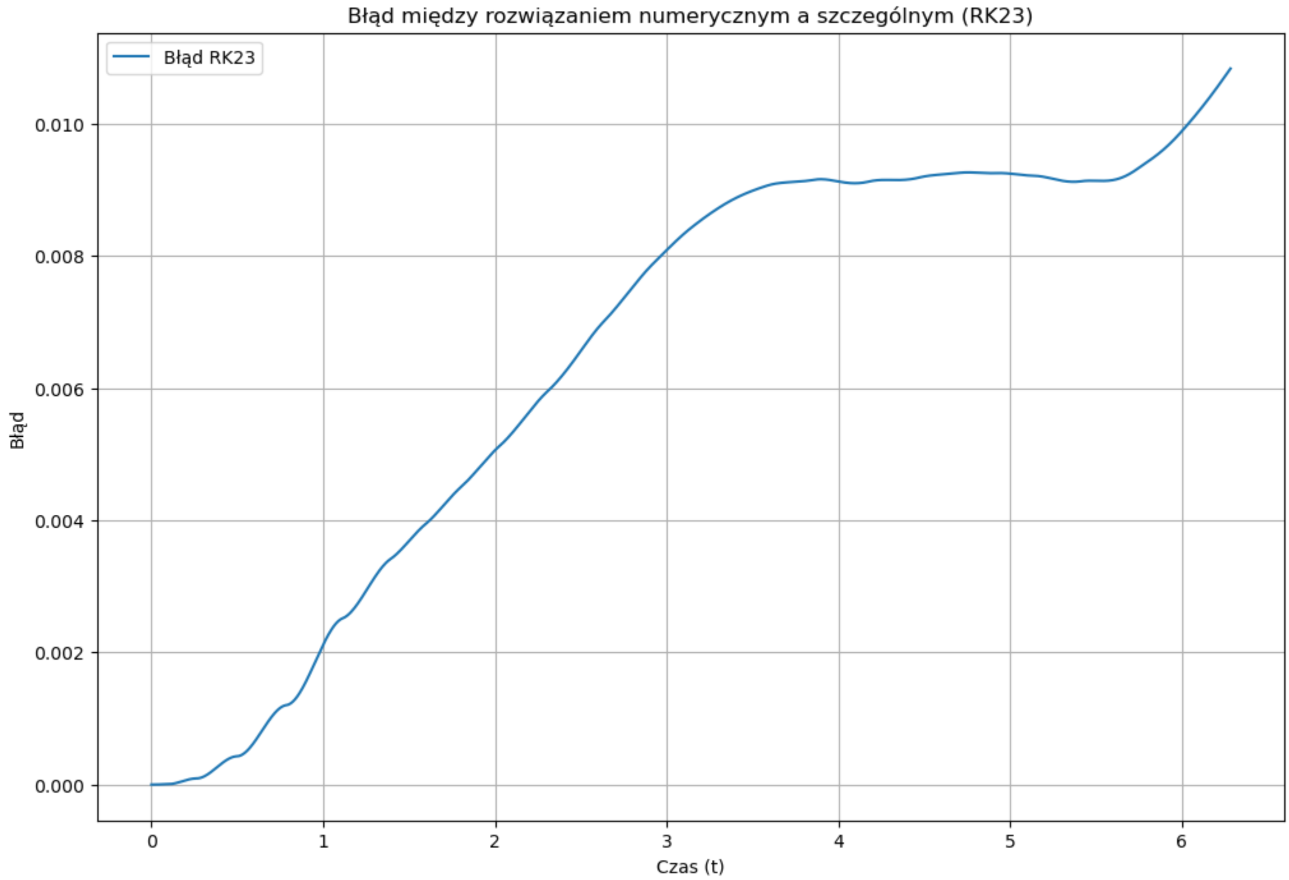
In [16]:
```python
from scipy.integrate import solve_ivp

# Zakres czasowy
t_span = (0, 2*np.pi)

# Rozwiązanie układu równań różniczkowych przy użyciu solve_ivp (RK45)
solution_rk45 = solve_ivp(system_equations, t_span, initial_conditions, method='RK45'

# Pobranie wartości z rozwiązania szczególnego
t_exact = np.linspace(0, 2*np.pi, 1000)
x_exact = np.array([[np.cos(ti), np.sin(ti), -np.sin(ti), np.cos(ti)] for ti in t_exa

# Obliczenie błędu między rozwiązaniem numerycznym a szczególnym
error_rk45 = np.linalg.norm(solution_rk45.sol(t_exact) - x_exact.T, axis=0)

# Rysowanie wykresu błędu
plt.figure(figsize=(12, 8))
plt.plot(t_exact, error_rk45, label="Błąd RK45")
plt.title("Błąd między rozwiązaniem numerycznym a szczególnym (RK45)")
plt.xlabel("Czas (t)")
plt.ylabel("Błąd")
plt.legend()
plt.grid(True)
plt.show()

# Ilość ocen funkcji (nfev) dla metody RK45
nfev_rk45 = solution_rk45.nfev
print(f"Ilość ocen funkcji (nfev) dla metody RK45: {nfev_rk45}")
```




```
Ilość ocen funkcji (nfev) dla metody RK45: 80
```

In [15]:

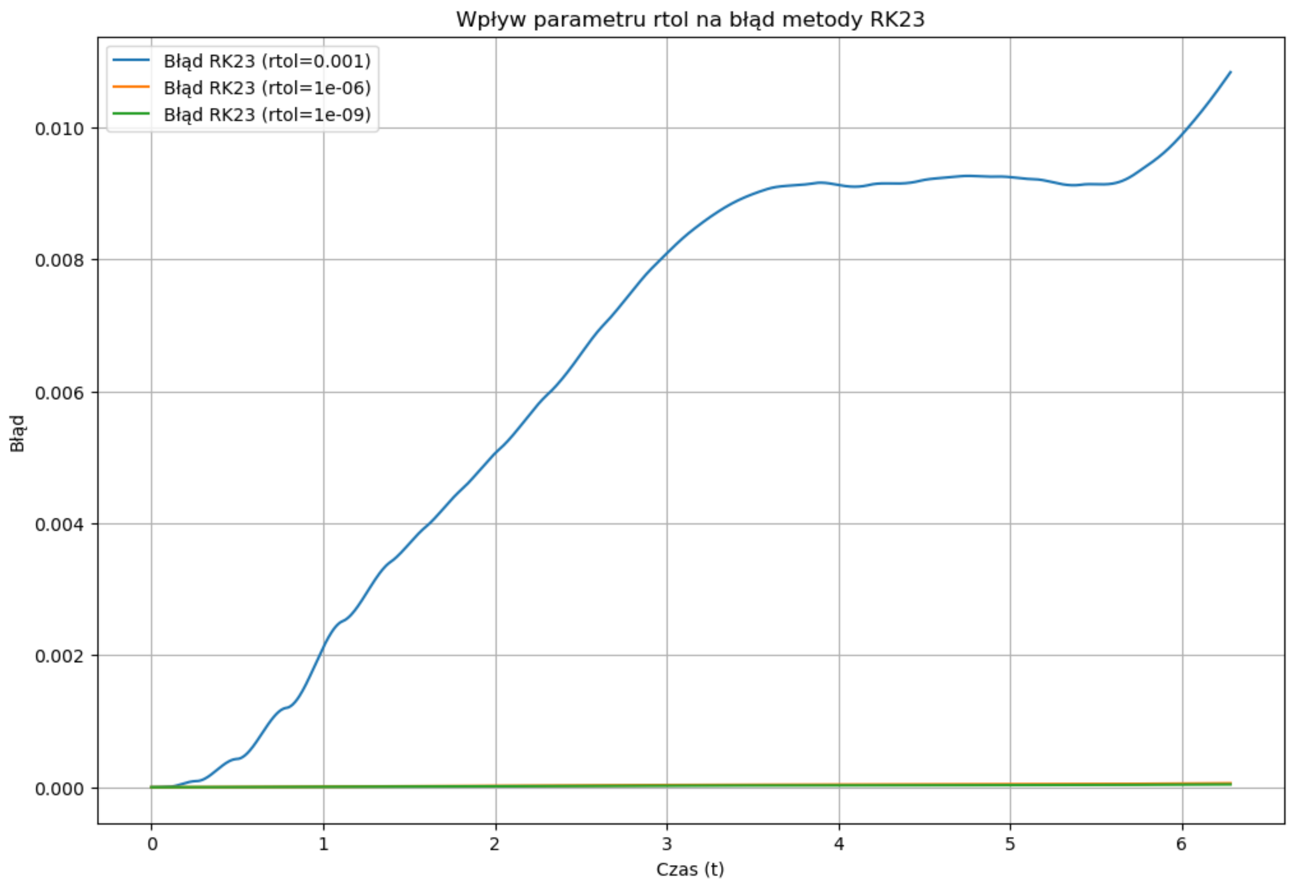
```python
# Rozwiązanie układu równań różniczkowych przy użyciu solve_ivp (RK23)
solution_rk23 = solve_ivp(system_equations, t_span, initial_conditions, method='RK23'

# Pobranie wartości z rozwiązania szczególnego
t_exact = np.linspace(0, 2*np.pi, 1000)
x_exact = np.array([[np.cos(ti), np.sin(ti), -np.sin(ti), np.cos(ti)] for ti in t_exa

# Obliczenie błędu między rozwiązaniem numerycznym a szczególnym
error_rk23 = np.linalg.norm(solution_rk23.sol(t_exact) - x_exact.T, axis=0)

# Rysowanie wykresu błędu
plt.figure(figsize=(12, 8))
plt.plot(t_exact, error_rk23, label="Błąd RK23")
plt.title("Błąd między rozwiązaniem numerycznym a szczególnym (RK23)")
plt.xlabel("Czas (t)")
plt.ylabel("Błąd")
plt.legend()
plt.grid(True)
plt.show()

# Ilość ocen funkcji (nfev) dla metody RK23
nfev_rk23 = solution_rk23.nfev
print(f"Ilość ocen funkcji (nfev) dla metody RK23: {nfev_rk23}")
```




```
Ilość ocen funkcji (nfev) dla metody RK23: 104
```

Porównując te dwie metody, błąd metody RK45 wyszedl 24,5 raza większy niż metody RK23

In [17]:
```
# Przetestujemy różne wartości rtol
rtol_values = [1e-3, 1e-6, 1e-9]

plt.figure(figsize=(12, 8))

for rtol in rtol_values:
    # Rozwiązanie układu równań różniczkowych przy użyciu solve_ivp (RK23)
    solution_rk23 = solve_ivp(system_equations, t_span, initial_conditions, method='R

    # Pobranie wartości z rozwiązania szczególnego
    t_exact = np.linspace(0, 2*np.pi, 1000)
    x_exact = np.array([[np.cos(ti), np.sin(ti), -np.sin(ti), np.cos(ti)] for ti in t_

    # Obliczenie błędu między rozwiązaniem numerycznym a szczególnym
    error_rk23 = np.linalg.norm(solution_rk23.sol(t_exact) - x_exact.T, axis=0)

    # Rysowanie wykresu błędu dla każdej wartości rtol
    plt.plot(t_exact, error_rk23, label=f"Błąd RK23 (rtol={rtol})")

plt.title("Wpływ parametru rtol na błąd metody RK23")
plt.xlabel("Czas (t)")
plt.ylabel("Błąd")
plt.legend()
plt.grid(True)
plt.show()
```



Wnioski: Parametr rtol w metodzie RK23 (Relative Tolerance) kontroluje tolerancję względną błędu numerycznego. Zwiększając rtol, zwiększamy tolerancję, co oznacza, że metoda będzie akceptować większy błąd w stosunku do wartości bezwzględnej aktualnego kroku czasowego.

Zadanie 5.

Orbita Arenstorfa. Jest to przykład z astronomii opisujący zredukowany problem trzech ciał. Rozważa się dwa ciała o masach μ i $\mu' = 1 - \mu$, poruszające się w ruchu kołowym na jednej płaszczyźnie oraz ciało o pomijalnej masie poruszające się między nimi w tej samej płaszczyźnie. Dany jest układ równań różniczkowych:

$$
\begin{aligned}
\dot{x_1}(t) &= x_2(t) \\
\dot{x_2}(t) &= x_1(t) + 2x_4(t) - \mu' \frac{x_1+\mu}{D_1} - \mu \frac{x_1-\mu'}{D_2} \\
\dot{x_3}(t) &= x_4(t) \\
\dot{x_4}(t) &= x_3(t) - 2x_2(t) - \mu' \frac{x_3(t)}{D_1} - \mu \frac{x_3(t)}{D_2}
\end{aligned}
.
$$

gdzie

$$
\begin{aligned}
D_1 &= ((x_1(t) + \mu)^2 + {x_3}^2(t))^{\frac{3}{2}} \\
D_2 &= ((x_1(t) - \mu')^2 + {x_3}^2(t))^{\frac{3}{2}} \\
\mu &= 0.012277471
\end{aligned}
.
$$

Zmienne x_1 i x_3 odpowiadają za współrzędne na płaszczyźnie trzeciego ciała zaś x_2 i x_4 są odpowiednio prędkościami. Warto zwrócić uwagę, że zarówno czas jak i masa zostały w równaniach przeskalowane, i nie mają bezpośredniej interpretacji fizycznej, należy je traktować jako zmienne bezwymiarowe. Dla pewnych warunków początkowych i czasu symulacji

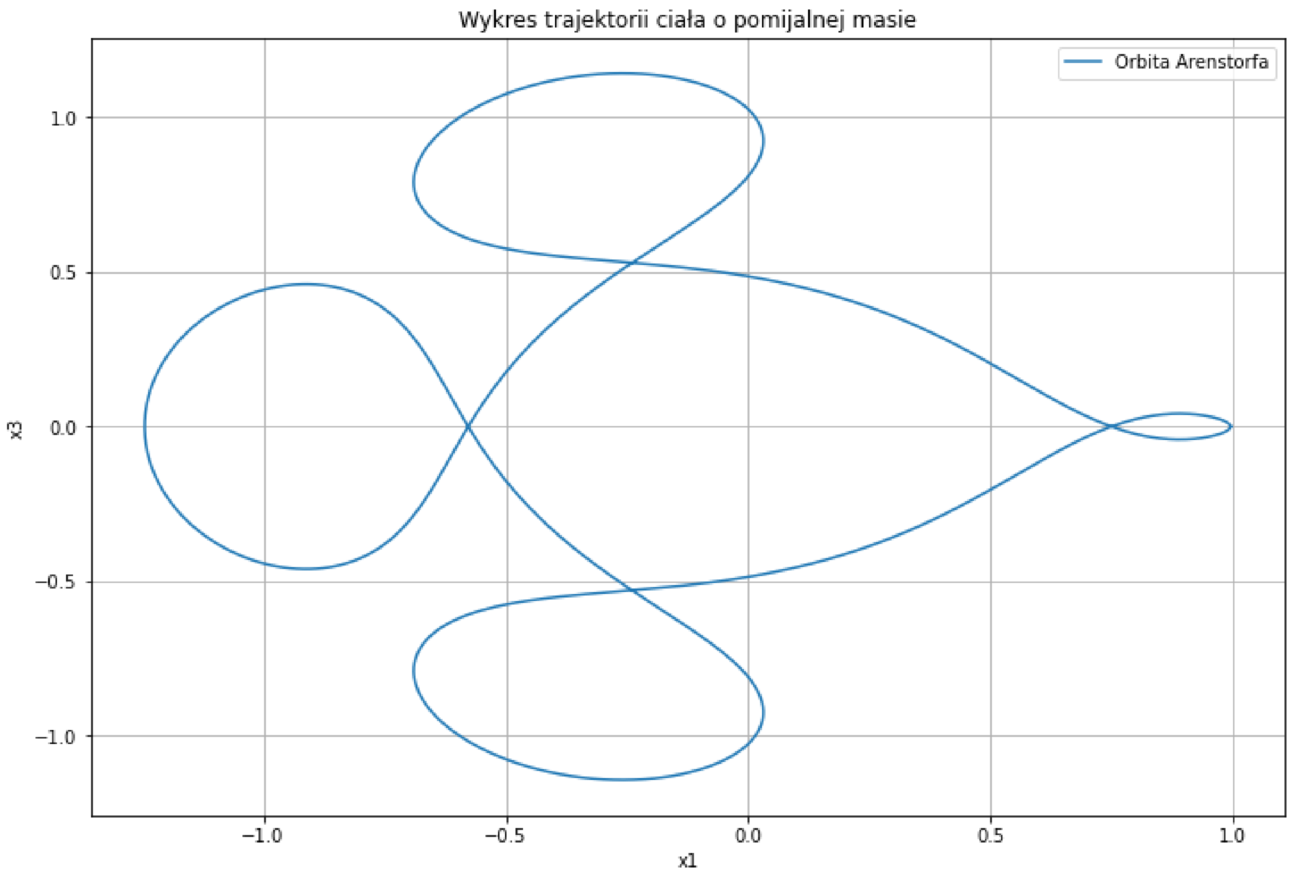
$$
\begin{aligned}
x_1(0) &= 0.994 \\
x_2(0) &= 0 \\
x_3(0) &= 0 \\
x_4(0) &= -2.00158510637908252240537862224 \\
T &= 17.0652165601579625588917206249.
\end{aligned}
$$

dokładne rozwiązanie tych równań jest okresowe ($x(0) = x(T)$).

Funkcję obliczającą pochodne zaimplementuj w main.py

Narysuj wykres uzyskanych rozwiązań numerycznych.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
def arenstorf_orbit(t, y):
    mu = 0.012277471
    mu_prime = 1 - mu

    x1, x2, x3, x4 = y
    D1 = ((x1 + mu)**2 + x3**2)**1.5
    D2 = ((x1 - mu_prime)**2 + x3**2)**1.5

    dx1dt = x2
    dx2dt = x1 + 2*x4 - mu_prime * (x1 + mu) / D1 - mu * (x1 - mu_prime) / D2
    dx3dt = x4
    dx4dt = x3 - 2*x2 - mu_prime * x3 / D1 - mu * x3 / D2

    return [dx1dt, dx2dt, dx3dt, dx4dt]

# Warunki początkowe
y0 = [0.994, 0, 0, -2.00158510637908252240537862224]
t_span = [0, 17.0652165601579625588917206249]

# Rozwiązanie układu równań różniczkowych
solution = solve_ivp(arenstorf_orbit, t_span, y0, method='RK45', rtol=1e-10, atol=1e-

# Wykres rozwiązania numerycznego
plt.figure(figsize=(12, 8))

# Wykres trajektorii ciała
plt.plot(solution.y[0], solution.y[2], label='Orbita Arenstorfa')
plt.xlabel('x1')
plt.ylabel('x3')
plt.title('Wykres trajektorii ciała o pomijalnej masie')
plt.legend()
plt.grid(True)
plt.show()
```



Bibliografia

1. J. C. Butcher. Numerical Methods for Ordinary Differential Equations. John Wiley and Sons, Ltd., 2003.
2. Z. Fortuna, B. Macukow, and J. Wąsowski. Metody numeryczne. WNT Warszawa, 1982.
3. E. Hairer, S.P. Nørsett, and G. Wanner. Solving Ordinary Differential Equations: I Nonstiff problems. Springer, 2 edition, 2000.
4. W. Mitkowski. Równania macierzowe i ich zastosowania. Wydawnictwa AGH, Kraków,2 edition, 2007.
5. A. Ralston. Wstęp do analizy numerycznej. PWN, Warszawa, 1965.
6. L. F. Shampine, I. Gladwell, and S. Thompson. Solving ODEs with MATLAB. Cambridge University Press, 2003.
7. Stoer, J., Burlirsch, R., 1980: Wstêp do metod numerycznych, tom 2. PWN Warszawa.

In [ ]: