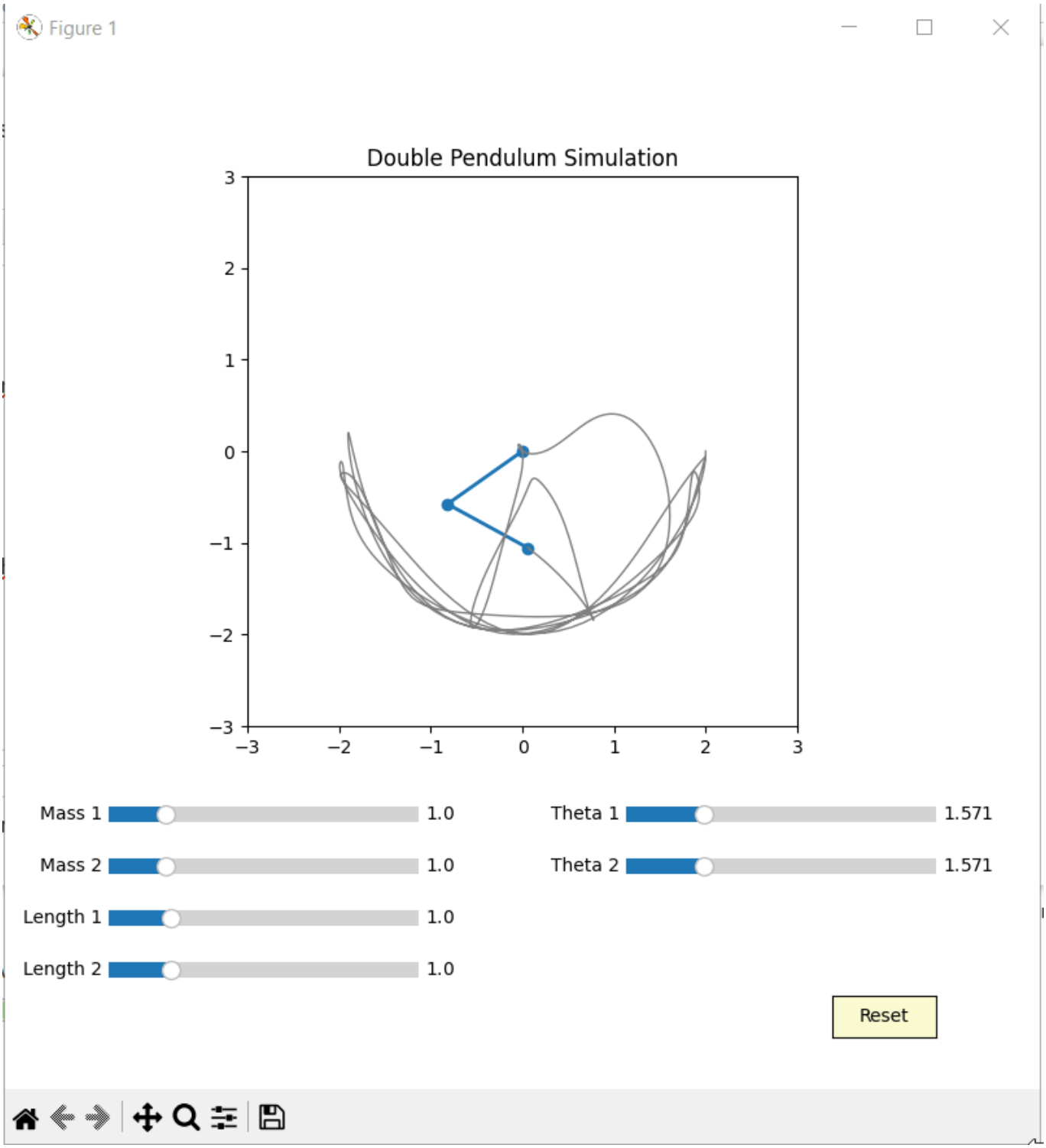
Python: Double pendulum simulator

PY: Symulator Wahadła Podwójnego



To nie jest zwykłe wahadło — to **wahadło podwójne** (ang. _double pendulum_), jeden z najprostszych układów fizycznych pokazujących zjawisko **deterministycznego chaosu**. Dwa ramiona, dwie masy, grawitacja — a efekt to piękny, nieprzewidywalny taniec, w którym drobna

zmiana kąta startowego potrafi całkowicie zmienić przyszły ruch.

Co tu się właściwie dzieje?

Układ składa się z dwóch punktowych mas zawieszonych na nierozciągliwych prętach. Ruch opisuje się czterema zmiennymi:

- \(\theta_1 \) – kąt pierwszego ramienia względem pionu,
- \(\theta_2 \) – kąt drugiego ramienia względem pionu,
- \(\omega_1 = \dot{\theta}_1 \) – prędkość kątowa pierwszego wahadła,
- \(\omega_2 = \dot{\theta}_2 \) – prędkość kątowa drugiego wahadła.

Równania ruchu pochodzą z zasad dynamiki Newtona albo bezpośrednio z mechaniki Lagrange'a:

$$ \begin{align*} \delta &= \theta_2 - \theta_1 \\ \ddot{\theta}_1 &= \frac{m_2 l_1 \omega_1^2 \sin\delta \cos\delta + m_2 g \sin\theta_2 \cos\delta + m_2 l_2 \omega_2^2 \sin\delta - (m_1 + m_2) g \sin\theta_1}{(m_1 + m_2) l_1 - m_2 l_1 \cos^2\delta} \\ \ddot{\theta}_2 &= \frac{-(m_1 + m_2) l_1 \omega_1^2 \sin\delta + (m_1 + m_2) g \sin\theta_1 \cos\delta - m_2 l_2 \omega_2^2 \sin\delta \cos\delta - (m_1 + m_2) g \sin\theta_2}{\left(\frac{l_2}{l_1} \right)((m_1 + m_2) l_1 - m_2 l_1 \cos^2\delta)} \end{align*} $$ Wygląda dziko? Tak właśnie wygląda fizyka nieliniowa :) Ruch symulowany jest numerycznie (metodą Rungego-Kutty) i animowany z pomocą biblioteki **matplotlib** w Pythonie. Kod pozwala interaktywnie zmieniać: - masy obu odważników, - długości ramion, - kąty początkowe. Każde kliknięcie to nowy dziwny świat trajektorii. ===== Matematyczne zaplecze: Runge-Kutta i mechanika Lagrange'a ===== Zanim komputer może cokolwiek zasymulować, potrzebujemy dwóch rzeczy: **modelu fizycznego** (czyli równań) i **metody ich rozwiązania**. Dla naszego podwójnego wahadła: - model opisuje **mechanika Lagrange'a**, - rozwiązanie daje **metoda Rungego-Kutty 4. rzędu (RK4)**. ---- === Mechanika Lagrange'a === Zamiast klasycznych sił z II zasady Newtona, Lagrange korzysta z zasad energii. Wprowadzamy funkcję **Lagrangian**: $$ L = T - V $$ Gdzie: - T — energia kinetyczna, - V — energia potencjalna.

Dla układu o współrzędnych uogólnionych q_i, równania ruchu wyprowadza się z tzw. **równań Lagrange'a**:

$$ \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{q}_i} \right) - \frac{\partial L}{\partial q_i} = 0 $$

W naszym przypadku: $q_1 = \theta_1$, $q_2 = \theta_2$ — kąty ramion.

Energie układu wyrażają się następująco:

- Energia kinetyczna:

$$ T = \frac{1}{2} m_1 l_1^2 \dot{\theta}_1^2 + \frac{1}{2} m_2 \left[l_1^2 \dot{\theta}_1^2 + l_2^2 \dot{\theta}_2^2 + 2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) \right] $$

- Energia potencjalna (w polu grawitacyjnym):

$$ V = - (m_1 + m_2) g l_1 \cos\theta_1 - m_2 g l_2 \cos\theta_2 $$

Podstawiamy do $L = T - V$, a następnie wstawiamy do równań Lagrange'a. W efekcie otrzymujemy dwa nieliniowe równania różniczkowe drugiego rzędu, które potem przekształcamy do układu równań

pierwszego rzędu.

Metoda Rungego-Kutty 4. rzędu (RK4)

Metoda RK4 pozwala numerycznie rozwiązać układ równań różniczkowych pierwszego rzędu:

$$ \dot{y} = f(t, y), \quad y(t_0) = y_0 $$

Aby znaleźć y_{n+1} w punkcie $t_{n+1} = t_n + h$, obliczamy:

$$ \begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f(t_n + h, y_n + h k_3) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} $$

To uśrednienie czterech oszacowań zmian stanu. Daje to dokładność lokalną rzędu $\mathcal{O}(h^5)$ i globalną rzędu $\mathcal{O}(h^4)$.

W naszym kodzie użyto `solve_ivp()` z biblioteki `scipy`, która domyślnie używa wersji **RK45** — adaptacyjnej metody z kontrolą błędu i dynamicznym doborem kroku.

Dlaczego to działa

Układ, który modelujemy: - jest **nieliniowy** (kąty i ich pochodne pojawiają się w funkcjach trygonometrycznych), - wykazuje **chaotyczne zachowanie** (olbrzymia wrażliwość na warunki początkowe), - **nie ma rozwiązań analitycznych** — tylko symulacja numeryczna pozwala go przeanalizować.

Mechanika Lagrange’a pozwala zbudować poprawny fizycznie model oparty o zasady zachowania, a metoda Rungego-Kutty pozwala ten model **skutecznie i dokładnie zasymulować** na komputerze.

Linki dla ciekawskich

- Mechanika Lagrange’a (EN)
- Wikipedia: Chaos deterministyczny
- Wikipedia: Double Pendulum
- Interaktywny model online (MyPhysicsLab)
- MyPhysicsLab — symulacje fizyczne online
- Wolfram Demonstrations: Classical Mechanics
- Lecture notes: Numerical Analysis, Oxford

Kod programu

double_pendulum.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib.widgets import Slider, Button
from scipy.integrate import solve_ivp

# Constants and default parameters
g = 9.81

# Default parameters for masses, lengths and initial angles (in
radians)
default_params = {
    'm1': 1.0,
    'm2': 1.0,
    'l1': 1.0,
    'l2': 1.0,
    'theta1': np.pi / 2,
    'theta2': np.pi / 2,
    'omega1': 0.0,
    'omega2': 0.0,
}

def double_pendulum_derivs(t, y, m1, m2, l1, l2):
    """Returns the derivatives for the double pendulum system.

    y = [theta1, theta2, omega1, omega2]
    """
    theta1, theta2, omega1, omega2 = y

    delta = theta2 - theta1

    denom1 = (m1 + m2) * l1 - m2 * l1 * np.cos(delta)**2
    domega1_dt = (m2 * l1 * omega1**2 * np.sin(delta) * np.cos(delta) +
                  m2 * g * np.sin(theta2) * np.cos(delta) +
                  m2 * l2 * omega2**2 * np.sin(delta) -
                  (m1 + m2) * g * np.sin(theta1)) / denom1

    denom2 = (l2 / l1) * denom1
    domega2_dt = (- m2 * l2 * omega2**2 * np.sin(delta) * np.cos(delta)
+
                  (m1 + m2) * g * np.sin(theta1) * np.cos(delta) -
                  (m1 + m2) * l1 * omega1**2 * np.sin(delta) -
                  (m1 + m2) * g * np.sin(theta2)) / denom2

    return [omega1, omega2, domega1_dt, domega2_dt]
```

```
def simulate(params, t_max=20, dt=0.02):
    """Simulate the double pendulum motion with given parameters."""
    t_span = (0, t_max)
    t_eval = np.arange(0, t_max, dt)
    y0 = [params['theta1'], params['theta2'], params['omega1'],
params['omega2']]
    sol = solve_ivp(double_pendulum_derivs, t_span, y0,
args=(params['m1'], params['m2'], params['l1'], params['l2']),
                    t_eval=t_eval, method='RK45')
    return sol.t, sol.y

# Initial simulation data
t, y = simulate(default_params)
theta1_vals = y[0]
theta2_vals = y[1]

def get_positions(theta1, theta2, l1, l2):
    """Calculate positions of pendulum bobs."""
    x1 = l1 * np.sin(theta1)
    y1 = -l1 * np.cos(theta1)
    x2 = x1 + l2 * np.sin(theta2)
    y2 = y1 - l2 * np.cos(theta2)
    return x1, y1, x2, y2

# Create the figure and the animation axes
fig, ax = plt.subplots(figsize=(8, 8))
plt.subplots_adjust(left=0.1, bottom=0.35)
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)
ax.set_aspect('equal')
ax.set_title('Double Pendulum Simulation')

# Initialize line and bob markers
line, = ax.plot([], [], 'o-', lw=2)
trace, = ax.plot([], [], '-', lw=1, color='gray')  # Optional trace of
second bob
trace_x, trace_y = [], []

def init():
    line.set_data([], [])
    trace.set_data([], [])
    return line, trace

# Animation update function
def update(frame):
    theta1 = theta1_vals[frame]
    theta2 = theta2_vals[frame]
    x1, y1, x2, y2 = get_positions(theta1, theta2,
current_params['l1'], current_params['l2'])
    line.set_data([0, x1, x2], [0, y1, y2])
    trace_x.append(x2)
```

```
    trace_y.append(y2)
    trace.set_data(trace_x, trace_y)
    return line, trace

# Create sliders for initial parameters
axcolor = 'lightgoldenrodyellow'
ax_m1 = plt.axes([0.1, 0.25, 0.3, 0.03], facecolor=axcolor)
ax_m2 = plt.axes([0.1, 0.20, 0.3, 0.03], facecolor=axcolor)
ax_l1 = plt.axes([0.1, 0.15, 0.3, 0.03], facecolor=axcolor)
ax_l2 = plt.axes([0.1, 0.10, 0.3, 0.03], facecolor=axcolor)
ax_theta1 = plt.axes([0.6, 0.25, 0.3, 0.03], facecolor=axcolor)
ax_theta2 = plt.axes([0.6, 0.20, 0.3, 0.03], facecolor=axcolor)

slider_m1 = Slider(ax_m1, 'Mass 1', 0.1, 5.0,
valinit=default_params['m1'])
slider_m2 = Slider(ax_m2, 'Mass 2', 0.1, 5.0,
valinit=default_params['m2'])
slider_l1 = Slider(ax_l1, 'Length 1', 0.5, 3.0,
valinit=default_params['l1'])
slider_l2 = Slider(ax_l2, 'Length 2', 0.5, 3.0,
valinit=default_params['l2'])
slider_theta1 = Slider(ax_theta1, 'Theta 1', 0, 2*np.pi,
valinit=default_params['theta1'])
slider_theta2 = Slider(ax_theta2, 'Theta 2', 0, 2*np.pi,
valinit=default_params['theta2'])

# Dictionary to hold current simulation parameters
current_params = default_params.copy()

def update_simulation(val):
    """Update simulation based on slider values."""
    global t, y, theta1_vals, theta2_vals, trace_x, trace_y,
current_params

    # Update current parameters from sliders
    current_params['m1'] = slider_m1.val
    current_params['m2'] = slider_m2.val
    current_params['l1'] = slider_l1.val
    current_params['l2'] = slider_l2.val
    current_params['theta1'] = slider_theta1.val
    current_params['theta2'] = slider_theta2.val
    current_params['omega1'] = 0.0
    current_params['omega2'] = 0.0

    # Re-run the simulation with new parameters
    t, y = simulate(current_params)
    theta1_vals = y[0]
    theta2_vals = y[1]

    # Clear the trace and reset animation frame index
    trace_x.clear()
```

```
    trace_y.clear()
    ani.frame_seq = ani.new_frame_seq()
    fig.canvas.draw_idle()

# Call update_simulation when any slider value changes
slider_m1.on_changed(update_simulation)
slider_m2.on_changed(update_simulation)
slider_l1.on_changed(update_simulation)
slider_l2.on_changed(update_simulation)
slider_theta1.on_changed(update_simulation)
slider_theta2.on_changed(update_simulation)

# Button to reset sliders to default values
reset_ax = plt.axes([0.8, 0.05, 0.1, 0.04])
button_reset = Button(reset_ax, 'Reset', color=axcolor,
hovercolor='0.975')

def reset(event):
    slider_m1.reset()
    slider_m2.reset()
    slider_l1.reset()
    slider_l2.reset()
    slider_theta1.reset()
    slider_theta2.reset()

button_reset.on_clicked(reset)

# Create the animation
ani = FuncAnimation(fig, update, frames=len(t), init_func=init,
interval=20, blit=True)

plt.show()
```

2025/05/07 13:30 · administrator