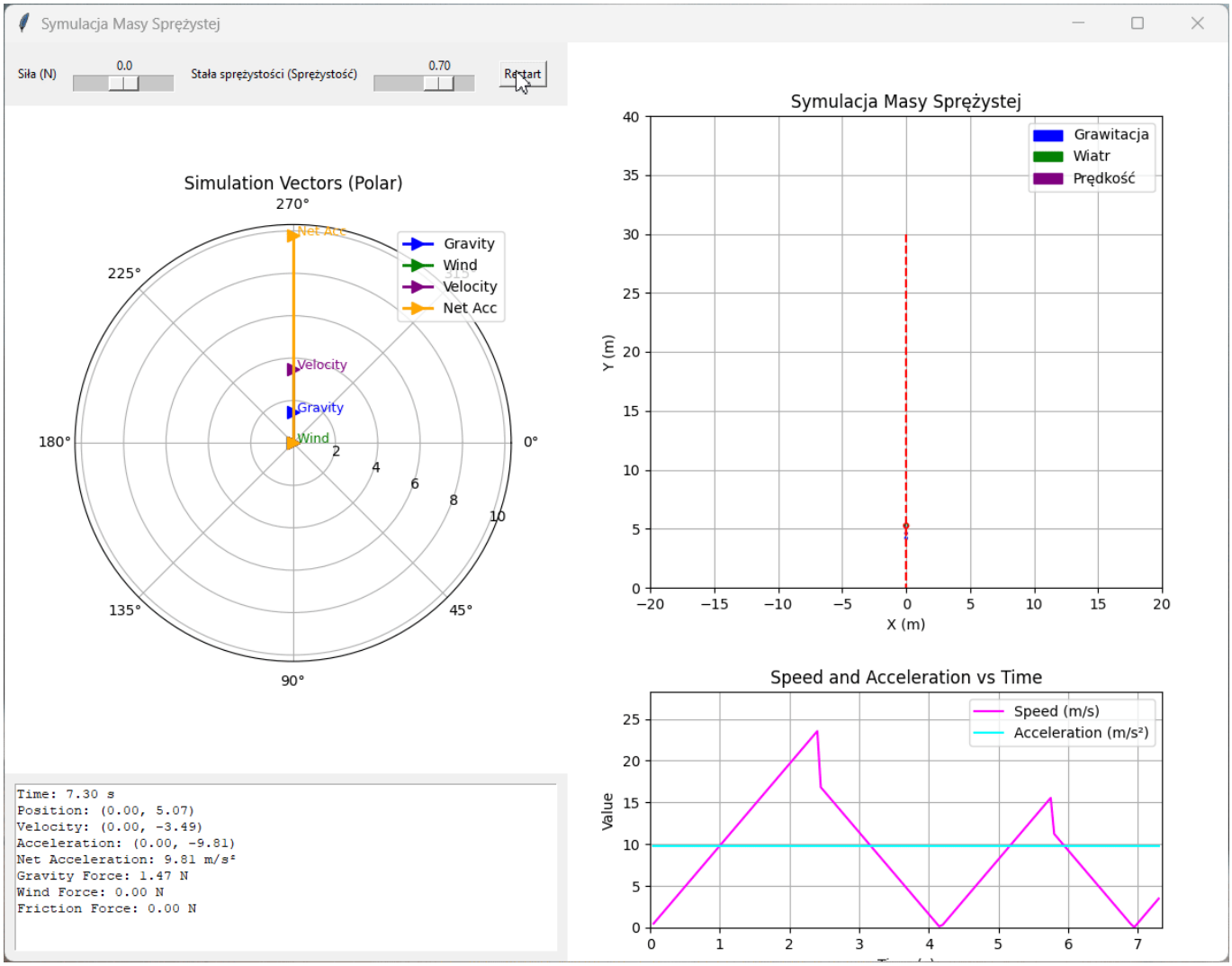
Python: Elastic body simulation

Downloadable script:

apple_sim.py



PY: Elastic Mass Simulation

This article explains how a script works to simulate the movement of an elastic mass under the influence of gravity, wind and friction, using the libraries `Tkinter`, `Matplotlib` i `NumPy`.

Physical constants and initial conditions

```
g = 9.81                # Przyspieszenie ziemskie (m/s^2)
m = 0.15                # Masa piłki (kg)
restitution = 0.7       # Współczynnik sprężystości (odbicia)
```

```
friction_coefficient = 0.5  # Współczynnik tarcia
```

We use basic constants from physics:

- Gravitational force: (F_g = m g)
- The rebound of the ball from the ground occurs with a reduction in vertical velocity by the coefficient of elasticity: $$ v_{y, \text{po}} = -v_{y, \text{przed}} \cdot e $$

Force and acceleration calculations

Code:

```
Fw = wind_force
ax_acc = Fw / m
ay_acc = -g
```

According to Newton's 2nd principle of dynamics:

- The wind force generates horizontal acceleration: $$ a_x = \frac{F_{\text{wiatr}}}{m} $$
- Vertical acceleration is equal to the Earth's acceleration: $$ a_y = -g $$

Kinematics of motion

```
vx += ax_acc * time_step
vy += ay_acc * time_step
x += vx * time_step + 0.5 * ax_acc * time_step**2
y += vy * time_step + 0.5 * ay_acc * time_step**2
```

We use the basic equations of kinematics:

$$ v = v_0 + a \cdot \Delta t \\ s = s_0 + v \cdot \Delta t + \frac{1}{2} a \cdot \Delta t^2 $$

Ground bounce and friction

```
if y <= y_min:
    vy = -vy * restitution
    if abs(vy) < 0.5:
        vx -= \dots \text{(siła tarcia)}
```

Kinetic friction force:

$$ F_{\text{tarcia}} = \mu \cdot m \cdot g \\ a_{\text{tarcia}} = \frac{F_{\text{tarcia}}}{m} = \mu \cdot g $$

Force vectors and diagrams

The code draws vectors:

- Gravity - always downwards
- Wind - horizontal
- Speed - dynamically

```
draw_arrow(x, y, vx * scale, vy * scale, 'purple')
```

Vectors in the polar diagram

```
draw_polar_arrow(polar_ax, angle, magnitude, color)
```

This section converts the force and velocity values to polar coordinates. This allows the user to observe their directions and relative values.

Graphs of speed and acceleration

```
current_speed = np.sqrt(vx**2 + vy**2)
net_acc = np.sqrt(ax_acc**2 + ay_acc**2)
```

These values are added to the lists and drawn in real time.

Summary

This script is an example of a dynamic simulation of motion in a two-dimensional force field, taking into account:

- Gravity
- Wind as an external force
- Elasticity on impact with the ground.
- Dynamic friction at low speed

Thanks to the graphical interface, the user can manipulate the parameters and observe the physical effects in real time.

Code

apple_sim.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.animation as animation
import tkinter as tk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

# Constants
g = 9.81              # Gravity (m/s^2)
m = 0.15              # Mass of the ball (kg)
restitution = 0.7     # Coefficient of restitution (bounciness)
time_step = 0.05      # Time step (s)
friction_coefficient = 0.5  # Friction coefficient between ball and
ground

# Graph boundaries (bigger graph)
x_min, x_max = -20, 20
y_min, y_max = 0, 40

# Scaling factor for the velocity arrow in simulation plot
velocity_scale = 0.2  # Adjust to control arrow length

# Global time and data lists for the secondary graph
sim_time = 0.0
time_list = []
speed_list = []
acc_list = []

# Initial Conditions
def reset_simulation():
    global x, y, vx, vy, trace, sim_time, time_list, speed_list,
acc_list
    x, y = 0, 30  # Start higher in the bigger graph
    vx, vy = 0, 0  # Reset velocities
    trace = []     # Reset trace
    sim_time = 0.0
    time_list = []
    speed_list = []
    acc_list = []

# Initialize wind force and spring constant
global wind_force
wind_force = 0.0
global spring_constant
spring_constant = 0.7

# Reset simulation at the beginning
reset_simulation()
```

```
# Tkinter Setup
root = tk.Tk()
root.title("Symulacja Masy Sprężystej")
# Set a larger window size to accommodate the controls and graphs
root.geometry("1200x900")

# Create two main frames: left for controls/info, right for graphs
left_frame = tk.Frame(root)
left_frame.pack(side="left", fill="both", expand=True)
right_frame = tk.Frame(root)
right_frame.pack(side="right", fill="both", expand=True)

def update_wind(val):
    global wind_force
    wind_force = float(val)

def update_spring(val):
    global restitution
    restitution = float(val)

# ------------------ Left Frame ------------------
# Create a frame for controls in the left_frame
control_frame = tk.Frame(left_frame)
control_frame.pack(side="top", fill="x", padx=10, pady=10)

# Create sliders for wind force and restitution in control_frame
tk.Label(control_frame, text="Siła (N)").pack(side="left")
wind_slider = tk.Scale(control_frame, from_=-2, to=2, resolution=0.1,
                       orient=tk.HORIZONTAL, command=update_wind)
wind_slider.pack(side="left", padx=10)

tk.Label(control_frame, text="Stała sprężystości
(Sprężystość)").pack(side="left")
spring_slider = tk.Scale(control_frame, from_=0, to=1, resolution=0.05,
                         orient=tk.HORIZONTAL, command=update_spring)
spring_slider.set(restitution)
spring_slider.pack(side="left", padx=10)

restart_button = tk.Button(control_frame, text="Restart",
command=reset_simulation)
restart_button.pack(side="left", padx=10)

# Create a text widget for real-time simulation values in left_frame
(below controls)
info_text = tk.Text(left_frame, height=10, width=50)
info_text.pack(side="bottom", fill="x", padx=10, pady=10)

# ------------------ Right Frame ------------------
# Create the simulation figure (main graph)
fig, ax = plt.subplots(figsize=(8, 6))
ax.set_xlim(x_min, x_max)
```

```
ax.set_ylim(y_min, y_max)
ax.set_xlabel("X (m)")
ax.set_ylabel("Y (m)")
ax.set_title("Symulacja Masy Sprężystej")
ax.grid()

apple = patches.Circle((0, 30), 0.2, color='red')
ax.add_patch(apple)

trace_line, = ax.plot([], [], 'r--', label='Trace')

def draw_arrow(start_x, start_y, dx, dy, color):
    return ax.arrow(start_x, start_y, dx, dy, head_width=0.2,
color=color)

# Initial force arrows
g_vector = draw_arrow(x, y, 0, -1, 'blue')
wind_vector = draw_arrow(x, y, wind_force, 0, 'green')
resultant_vector = draw_arrow(x, y, vx * velocity_scale, vy *
velocity_scale, 'purple')
ax.legend([g_vector, wind_vector, resultant_vector], ["Grawitacja",
"Wiatr", "Prędkość"])

# Embed the simulation figure in the right_frame
sim_canvas = FigureCanvasTkAgg(fig, master=right_frame)
sim_canvas.get_tk_widget().pack(side="top", fill="both", expand=True)

# Create the secondary graph for speed and acceleration
fig2, ax2 = plt.subplots(figsize=(5, 3))
ax2.set_xlabel("Time (s)")
ax2.set_ylabel("Value")
ax2.set_title("Speed and Acceleration vs Time")
ax2.grid()
line_speed, = ax2.plot([], [], label="Speed (m/s)", color='magenta')
line_acc, = ax2.plot([], [], label="Acceleration (m/s²)", color='cyan')
ax2.legend()

# Embed the secondary graph in the right_frame below the simulation
graph
data_canvas = FigureCanvasTkAgg(fig2, master=right_frame)
data_canvas.get_tk_widget().pack(side="top", fill="both", expand=True)

# Create the polar graph for simulation vectors
polar_fig, polar_ax = plt.subplots(figsize=(5, 3),
subplot_kw={'projection': 'polar'})
polar_ax.set_title("Simulation Vectors (Polar)")
# Embed the polar graph in the right_frame below the other graphs
polar_canvas = FigureCanvasTkAgg(polar_fig, master=left_frame)
polar_canvas.get_tk_widget().pack(side="top", fill="both", expand=True)

# Helper function to draw an arrow in the polar plot
```

```
def draw_polar_arrow(ax, angle, r, color, label=None):
    # Draw a line from (angle, 0) to (angle, r) with an arrow marker.
    ax.plot([angle, angle], [0, r], color=color, linewidth=2,
marker='>', markersize=8, label=label)
    # Optionally, add a text label near the tip:
    ax.text(angle, r, f" {label}", color=color, fontsize=9)

# ------------------ Animation Function ------------------
def animate(i):
    global x, y, vx, vy, wind_force, g_vector, wind_vector,
resultant_vector
    global trace, sim_time, time_list, speed_list, acc_list

    # Forces
    Fw = wind_force  # Wind force (N)

    # Compute accelerations (without friction yet)
    ax_acc = Fw / m      # Horizontal acceleration due to wind
    ay_acc = -g          # Vertical acceleration (gravity)

    # Calculate net acceleration magnitude and its angle
    net_acc = np.sqrt(ax_acc**2 + ay_acc**2)
    net_acc_angle = np.arctan2(ay_acc, ax_acc)  # angle of net
acceleration

    # Update velocities (before friction)
    vx += ax_acc * time_step
    vy += ay_acc * time_step

    # Update positions using kinematics
    x += vx * time_step + 0.5 * ax_acc * time_step**2
    y += vy * time_step + 0.5 * ay_acc * time_step**2

    # Bounce off the ground and apply friction if vertical speed is low
    friction_force = 0
    friction_angle = 0
    if y <= y_min:
        y = y_min
        vy = -vy * restitution

        # Apply friction if vertical motion is small (ball nearly at
rest on ground)
        if abs(vy) < 0.5:
            friction_acc = friction_coefficient * g
            friction_force = friction_coefficient * m * g  # friction
force magnitude
            if vx > 0:
                vx = max(vx - friction_acc * time_step, 0)
            elif vx < 0:
                vx = min(vx + friction_acc * time_step, 0)
            # Friction opposes the velocity direction
```

```
            velocity_angle = np.arctan2(vy, vx) if (vx != 0 or vy != 0)
else 0
            friction_angle = (velocity_angle + np.pi) % (2 * np.pi)

    # Bounce off the sides
    if x <= x_min:
        x = x_min
        vx = -vx * restitution
    elif x >= x_max:
        x = x_max
        vx = -vx * restitution

    # Update apple position
    apple.set_center((x, y))

    # Update trace
    trace.append((x, y))
    trace_line.set_data(*zip(*trace))

    # Remove and redraw force vectors in simulation plot
    g_vector.remove()
    wind_vector.remove()
    resultant_vector.remove()
    g_vector = draw_arrow(x, y, 0, -1, 'blue')
    wind_vector = draw_arrow(x, y, wind_force, 0, 'green')
    resultant_vector = draw_arrow(x, y, vx * velocity_scale, vy *
velocity_scale, 'purple')

    # Update simulation time and secondary graph data
    sim_time += time_step
    current_speed = np.sqrt(vx**2 + vy**2)
    time_list.append(sim_time)
    speed_list.append(current_speed)
    acc_list.append(net_acc)

    # Update secondary graph lines
    line_speed.set_data(time_list, speed_list)
    line_acc.set_data(time_list, acc_list)
    ax2.set_xlim(0, sim_time + time_step)
    ax2.set_ylim(0, max(max(speed_list, default=1), max(acc_list,
default=1)) * 1.2)
    data_canvas.draw()

    # Update text widget with real-time simulation values
    info_text.delete("1.0", tk.END)
    info_str = (
        f"Time: {sim_time:.2f} s\n"
        f"Position: ({x:.2f}, {y:.2f})\n"
        f"Velocity: ({vx:.2f}, {vy:.2f})\n"
        f"Acceleration: ({ax_acc:.2f}, {ay_acc:.2f})\n"
        f"Net Acceleration: {net_acc:.2f} m/s²\n"
```

```
        f"Gravity Force: {m * g:.2f} N\n"
        f"Wind Force: {wind_force:.2f} N\n"
        f"Friction Force: {friction_force:.2f} N\n"
    )
    info_text.insert(tk.END, info_str)

    # ------------------ Update Polar Graph ------------------
    # Compute polar vector parameters:
    # Gravity vector (force) - always downward (angle = -pi/2)
    gravity_magnitude = m * g
    gravity_angle = -np.pi/2  # or equivalently 3pi/2

    # Wind vector (force)
    wind_magnitude = abs(wind_force)
    wind_angle = 0 if wind_force >= 0 else np.pi

    # Velocity vector
    velocity_magnitude = np.sqrt(vx**2 + vy**2)
    velocity_angle = np.arctan2(vy, vx)

    # Clear and update polar axis
    polar_ax.clear()
    polar_ax.set_title("Simulation Vectors (Polar)")
    # Set 0° to the right and angles increasing clockwise
    polar_ax.set_theta_zero_location("E")
    polar_ax.set_theta_direction(-1)

    # Draw arrows in the polar plot
    draw_polar_arrow(polar_ax, gravity_angle, gravity_magnitude,
'blue', 'Gravity')
    draw_polar_arrow(polar_ax, wind_angle, wind_magnitude, 'green',
'Wind')
    draw_polar_arrow(polar_ax, velocity_angle, velocity_magnitude,
'purple', 'Velocity')
    draw_polar_arrow(polar_ax, net_acc_angle, net_acc, 'orange', 'Net
Acc')
    if friction_force > 0:
        draw_polar_arrow(polar_ax, friction_angle, friction_force,
'red', 'Friction')

    polar_ax.legend()
    polar_canvas.draw()

    return apple, g_vector, wind_vector, resultant_vector, trace_line,
line_speed, line_acc

def on_closing():
    root.quit()  # Stop the Tkinter event loop
    root.destroy()  # Destroy the Tkinter window

ani = animation.FuncAnimation(fig, animate, frames=300, interval=50,
```

```
blit=False)

# Handle window close event
root.protocol("WM_DELETE_WINDOW", on_closing)

root.mainloop()
```

2026/03/11 14:31 · administrator