= Symulacja Masy Sprężystej =

Ten artykuł opisuje działanie skryptu w Pythonie wykonującego symulację ruchu kulki pod wpływem siły grawitacji, wiatru oraz oddziaływania sprężysto-tarciowego. Skrypt wykorzystuje biblioteki NumPy, Matplotlib oraz Tkinter do wizualizacji i interakcji.

Spis treści

1. Opis ogólny

Skrypt emuluje ruch kulki („jabłuszka”) o masie m poruszającej się w układzie dwuwymiarowym. Użytkownik może regulować siłę wiatru oraz współczynnik sprężystości odbicia (restitution), a także obserwować: * Tor ruchu * Wektory sił (grawitacja, wiatr, prędkość, wyniku) * Wykres prędkości i przyspieszenia w funkcji czasu * Reprezentację wektorów w układzie polarnym

2. Stałe i parametry

<syntaxhighlight lang=„python”> # Stałe fizyczne g = 9.81 # przyspieszenie grawitacyjne [m/s²] m = 0.15 # masa kulki [kg] e = 0.7 # współczynnik sprężystości (restitution) μ = 0.5 # współczynnik tarcia dt = 0.05 # krok czasowy [s] </syntaxhighlight>

Parametry wykresów: <syntaxhighlight lang=„python”> x_min, x_max = -20, 20 # granice osi X [m] y_min, y_max = 0, 40 # granice osi Y [m] velocity_scale = 0.2 # skala strzałki prędkości </syntaxhighlight>

3. Model fizyczny

3.1 Przyspieszenie

Przyspieszenie kulki w kierunku poziomym i pionowym wyznaczamy z II zasady dynamiki Newtona: $a_x = \frac{F_\text{wiatr}}{m}, \quad a_y = -g$

3.2 Równania kinetyczne

Pozycję i prędkość w kolejnych krokach obliczamy ze wzorów: $v(t+\Delta t) = v(t) + a \,\Delta t$ $x(t+\Delta t) = x(t) + v(t)\,\Delta t + \tfrac12\,a\,\Delta t^2$

3.3 Odbicie od podłoża

Gdy kulka dotyka podłoża ($y \le y_\min$), następuje odbicie: $v_y' = -\,e\, v_y$ gdzie e to współczynnik sprężystości. Jeżeli prędkość pionowa jest mała ($|v_y|<0.5$), dodajemy tarcie:

$F_\text{tarcie} = \mu\, m\, g$ które zmniejsza prędkość poziomą: $v_x \;\to\; v_x \;-\;\mathrm{sign}(v_x)\,\mu\,g\,\Delta t$

4. Inicjalizacja i reset symulacji

Funkcja `reset_simulation()` przywraca początki symulacji: <syntaxhighlight lang=„python”> def reset_simulation():

```
  global x, y, vx, vy, trace, sim_time, time_list, speed_list, acc_list
  x, y = 0, 30         # początkowa pozycja
  vx, vy = 0, 0        # początkowe prędkości
  trace = []           # czyszczenie śladu toru
  sim_time = 0.0
  time_list = []
  speed_list = []
  acc_list = []
```

</syntaxhighlight>

5. Interfejs użytkownika (Tkinter)

Symulacja działa w oknie Tkinter, w którym mamy: * Suwak do zmiany siły wiatru (`wind_slider`) * Suwak do zmiany współczynnika sprężystości (`spring_slider`) * Przycisk „Restart” – ponowne wywołanie `reset_simulation()` * Pole tekstowe (`info_text`) wyświetlające bieżące wartości czasu, pozycji, prędkości, przyspieszenia i sił.

6. Rysowanie i animacja (Matplotlib)

Tworzymy trzy wykresy: # Główny wykres 2D: pozycja kulki i wektory sił # Wykres czasowy prędkości i przyspieszenia # Wykres polarny wektorów

Przykład inicjalizacji głównego wykresu: <syntaxhighlight lang=„python”> 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)”) apple = patches.Circle[1)]

[1)]
0, 30), 0.2, color='red') ax.add_patch(apple) trace_line, = ax.plot([], [], 'r–', label='Trace') </syntaxhighlight> == 7. Funkcja animate() == To serce animacji wywoływane przez `FuncAnimation`. W kolejnych krokach: <syntaxhighlight lang=„python”> def animate(i):

```
  global x, y, vx, vy, sim_time
```

```
  # 1) Obliczanie przyspieszeń
  ax_acc = wind_force / m
```

```
  ay_acc = -g
```

```
  # 2) Aktualizacja prędkości
  vx += ax_acc * dt
  vy += ay_acc * dt
```

```
  # 3) Aktualizacja pozycji
  x  += vx * dt + 0.5 * ax_acc * dt**2
  y  += vy * dt + 0.5 * ay_acc * dt**2
```

```
  # 4) Odbicie i tarcie
  if y <= y_min:
      y = y_min
      vy = -vy * restitution
      # ... kod tarcia ...
```

```
  # 5) Aktualizacja obiektu graficznego
  apple.set_center((x, y))
  trace.append((x, y))
  trace_line.set_data(*zip(*trace))
```

```
  # 6) Rysowanie wektorów na wykresie:
  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')
```

```
  # 7) Aktualizacja wykresów prędkości i przyspieszenia
  sim_time += dt
  current_speed = np.hypot(vx, vy)
  time_list.append(sim_time); speed_list.append(current_speed);
acc_list.append(np.hypot(ax_acc, ay_acc))
  line_speed.set_data(time_list, speed_list)
  line_acc.set_data(time_list, acc_list)
  ax2.set_xlim(0, sim_time+dt)
  ax2.set_ylim(0, max(max(speed_list,1), max(acc_list,1))*1.2)
  data_canvas.draw()
```

```
  # 8) Aktualizacja pola tekstowego
  info_text.delete("1.0", tk.END)
  info_text.insert(tk.END, f"Time: {sim_time:.2f} s\nPosition:
({x:.2f},{y:.2f})\n…")
```

```
  # 9) Wykres polarny
  polar_ax.clear()
  draw_polar_arrow(polar_ax, -np.pi/2, m*g, 'blue', 'Grawitacja')
  # … pozostałe wektory …
```

```
  polar_canvas.draw()
```

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

</syntaxhighlight> === 7.1 Uwaga nt. funkcji draw_polar_arrow === Funkcja rysuje wektor w układzie polarnym z określonym kątem θ i długością r, dodając znacznik strzałki oraz tekstową etykietę. == 8. Zakończenie i pętla główna == Na końcu tworzymy animację: <syntaxhighlight lang=„python”> ani = animation.FuncAnimation(fig, animate, frames=300, interval=50, blit=False) root.protocol(„WM_DELETE_WINDOW”, on_closing) root.mainloop() </syntaxhighlight> Funkcja `on_closing()` zapewnia poprawne zakończenie pętli Tkinter i zamknięcie okna. — Ten artykuł pokrywa wszystkie główne elementy skryptu: od definicji parametrów fizycznych, przez mechanikę ruchu, aż po elementy GUI i animacji. Dzięki zastosowaniu LaTeX-owych wzorów równania są czytelne i wiernie odzwierciedlają użyte w kodzie prawa dynamiki.