Wstęp do Modelowania Komputerowego
Chaotic pendulum

Gabriela Białoskórska, Mikołaj Knysak, Ignacy Tekieli
Fizyka Techniczna

Listopad 2020

Spis treści

1 Wprowadzenie

1.1 Cel pracy

Celem niniejszego projektu było utworzenie symulacji komputerowej, ukazującej jej użytkownikowi ruch wahadła podwójnego.

1.2 Wstęp Teoretyczny

1.2.1 Modelowanie komputerowe

Modelowanie komputerowe jest obecnie najbardziej efektywnym narzędziem analizy złożonych problemów nauki i techniki. Polega ono na stworzeniu symulacji - na pojedynczym komputerze lub kilku komputerach połączonych w sieci, w celu odtworzenia pożądanego zjawiska. Do tego celu wykorzystywany jest model abstrakcyjny (komputerowy, obliczeniowy).

Podstawową zaletą tej metody jest duża dokładność uzyskiwanych wyników oraz możliwość dynamicznej zmiany parametrów fizycznych modelu lub warunków jego pracy. Dzięki temu nie ma konieczności powtórnego konstruowania wirtualnej reprezentacji analizowanego zjawiska lub obiektu.

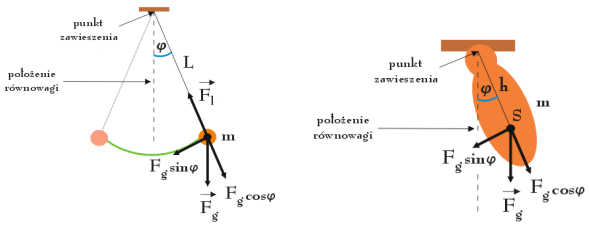
Precyzyjne modelowanie zjawisk fizycznych jest możliwe za pomocą skomplikowanego opisu matematycznego. Rozwiązanie go jest dostępne jedynie na drodze numerycznej. Z uwagi na to, stosowanie dokładnych modeli symulacyjnych niejednokrotnie wymaga ogromnych nakładów obliczeniowych, co może przekraczać możliwości sprzętowe standardowych komputerów osobistych. We wspomnianym przypadku do przeprowadzenia symulacji stosuje się zazwyczaj tzw. Superkomputery, które swoją mocą obliczeniową znacznie przewyższają te stosowane powszechnie na własny użytek.

Prezentowany przez nas projekt nie wymagał dużych nakładów obliczeniowych, ze względu na stosunkowo prosty model matematyczny symulowanego zjawiska, jakim jest ruch wahadła.

1.2.2 Wahadło

Wahadło to najprostszy układ nieliniowy, który może służyć do opisu zjawisk fizycznych. Zazwyczaj definiowany jest jako wyidealizowane ciało zawieszone w jednorodnym polu grawitacyjnym, wykonujące drgania wokół poziomej osi (nie przechodzącej przez środek ciężkości tego ciała).

W mechanice wyróżnia się dwa podstawowe modele wahadeł: matematyczne oraz fizyczne. Pojęcie wahadła matematycznego (prostego) odnosi się do punktu materialnego, zawieszonego na nieważkiej nici, natomiast w przypadku wahadła fizycznego mamy do czynienia z bryłą sztywną.



(a) Wahadło matematyczne (b) Wahadło fizyczne

Rysunek 1: Porównanie modeli wahadeł w mechanice

Dla wahadła matematycznego możemy zapisać następujące równania:

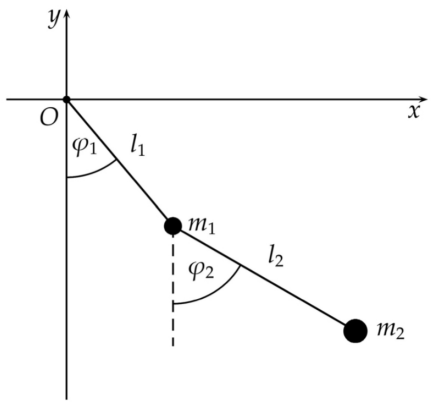
$$F = -mgsin\theta$$

$$F = -mgsin\theta = -\frac{mg}{l}x$$

$$T = 2\pi\sqrt{\frac{m}{k}} = 2\pi\sqrt{\frac{l}{g}}$$

1.2.3 Wahadło podwójne

Wahadłem matematycznym podwójnym nazywa się układ mechaniczny o dwóch stopniach swobody, znajdujący się w stacjonarnym jednorodnym polu siły ciężkości o przyspieszeniu g. Jest on złożony z dwóch wahadeł jednokrotnych o masach m_1 i m_2, skupionych na końcach wahadeł o długościach l_1 i l_2, z których drugie jest zamocowane przegubowo do pierwszego. Szkic geometryczny układu przesdtawia rys. 2.



Rysunek 2: Schematyczne przedstawienie wahadła podwójnego

Ruch wahadła odbywa się w ustalonej płaszczyźnie, przechodzącej przez linię pionu. φ_1 i φ_2 to kąty odchyleń wahadeł od pionu. Równania ruchu wahadła można znaleźć wykorzystując formalizm lagranżowski. Wobec tego:

Współrzędne i prędkości zawieszonych punktów:

$$(x_1, y_1) = (l_1 sin\varphi_1, -l_1 cos\varphi_1)$$

$$(x_2, y_2) = (l_1 sin\varphi_1 + l_2 sin\varphi_2, -l_1 cos\varphi_1 - l_2 cos\varphi_2)$$

$$(\dot{x}_1, \dot{y}_1) = (l_1\dot{\varphi}_1 cos\varphi_1, l_1\dot{\varphi}_1 sin\varphi_1)$$

$$(\dot{x}_2, \dot{y}_2) = (l_1\dot{\varphi}_1 cos\varphi_1 + l_2\dot{\varphi}_2 cos\varphi_2, l_1\dot{\varphi}_1 sin\varphi_1 + l_2\dot{\varphi}_2 sin\varphi_2)$$

Energia kinetyczna T i potencjalna U wahadła

$$T = \frac{1}{2}m_1 {v_1}^2 + \frac{1}{2}m_2 {v_2}^2 =$$

$$= \frac{1}{2}m_1({\dot{x_1}}^2 + {\dot{y_1}}^2) + \frac{1}{2}m_2({\dot{x_2}}^2 + {\dot{y_2}}^2) =$$

$$= \frac{1}{2}m_1 {l_1}^2 {\dot{\varphi_1}}^2 + \frac{1}{2}m_2({l_1}^2{\dot{\varphi_1}}^2 + {l_2}^2{\dot{\varphi_2}}^2 + 2l_1 l_2\dot{\varphi}_1\dot{\varphi}_2 cos(\varphi_1 - \varphi_2)) =$$

$$= \frac{1}{2}(m_1 + m_2){l_1}^2{\dot{\varphi_1}}^2 + \frac{1}{2}m_2{l_2}^2{\dot{\varphi_2}}^2 + m_2 l_1 l_2\dot{\varphi}_1^2\dot{\varphi}_2^2 cos(\varphi_1 - \varphi_2)$$

$$U = m_1 g y_1 + m_2 g y_2 = -(m_1 + m_2)g l_1 cos\varphi_1 - m_2 g l_2 cos\varphi_2$$

Na podstawie powyższych równań obliczono lagrangian układu **L = T - U**. Wykorzystując równania Lagrange'a

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{q}_j}) - \frac{\partial L}{\partial q_j} = 0,$$

$$j = 1, 2$$

Znaleziono **równania ruchu wahadła podwójnego**

$$\ddot{\varphi_1} = \frac{\frac{\mu}{\lambda}\dot{\varphi_2}^2 sin(\varphi_1 - \varphi_2) + \frac{g}{l_1} sin\varphi_1 + \frac{1}{2}\mu sin(2(\varphi_1 - \varphi_2))\dot{\varphi_1}^2 - \mu\frac{g}{l_1} sin\varphi_2 cos(\varphi_1 - \varphi_2)}{\mu cos^2(\varphi_1 - \varphi_2) - 1} \tag{1}$$

$$\ddot{\varphi_2} = \frac{\frac{1}{2}\mu sin(2(\varphi_1 - \varphi_2))\dot{\varphi_2}^2 + \frac{g}{l_2} sin\varphi_1 cos(\varphi_1 - \varphi_2) + \lambda\dot{\varphi_1}^2 sin(\varphi_1 - \varphi_2) - \frac{g}{l_2} sin\varphi_2}{1 - \mu cos^2(\varphi_1 - \varphi_2)} \tag{2}$$

gdzie:

$$\mu = \frac{m_2}{(m_1 + m_2)}$$

$$\lambda = \frac{l_1}{l_2}$$

Równania (1) i (2) tworzą układ dwóch sprzężonych nieliniowych równań zwyczajnych drugiego rzędu.

2 Materiały i Metody

2.1 Wiadomości wstępne

W celu stworzenia symulacji poruszania się opisanego we wstępie wahadła podwójnego, wykorzystaliśmy język Python. Jest to język programowania wysokiego poziomu, który cechuje się czytelnością i klarownością kodu źródłowego oraz składni. Jego szerokie spektrum zastosowania wynika z mnogości już istniejących bibliotek, a także łatwości w tworzeniu własnych.

Wyjątkowo pomocna w tworzeniu naszego projektu była biblioteka **Pygame**. Została ona stworzona przez Pete Shinnersa i służy do sterowania multimediami. Do działania wymaga biblioteki SDL, dzięki której dostarcza modułów pozwalających na wyświetlanie grafiki, odtwarzanie dźwięków, śledzenie czasu, obsługę myszy, joysticka i CD, czy też renderowanie czcionek TTF. Jest ona wieloplatformowa i umożliwia pracę na różnych systemach operacyjnych. Dystrybuowana na zasadach licencji LGPL.

2.2 Działanie programu

Działanie programu jest bardzo intuicyjne. Symulacja opiera się wyłącznie na wyżej wyprowadzonych równaniach ruchu wahadła podwójnego (1) i (2) oraz danych wprowadzonych przez użytkownika do konsoli. Całość poszczególnych fragmentów kodu (przedstawionych poniżej) jest kontrolowana i wywoływana w pętli controllerem.

Zdecydowaliśmy się na wykorzystanie wspomnianej we wstępie biblioteki Pygame, z uwagi na prostotę tworzonego programu. Nie wymagał on od nas generowania wizualizacji 3D, a jedynie pobierania danych i tworzenia na ich podstawie animacji trajektorii.

```python
#!/usr/bin/env python3
import argparse
from simulation import Simulation

if __name__ == '__main__':
    # Initializing args parser
    parser = argparse.ArgumentParser(description='Double pendulum simulator')

    # Adding arguments to parser
    parser.add_argument('--res', nargs=2, type=int, required=False, default=(800, 800),
                        help='Resolution of the simulation window. [px] Enter in format: --res x y')
    parser.add_argument('--fps', type=int, required=False, default=60,
                        help='Frame rate of the simulation. [fps]')
    parser.add_argument('--time', type=int, required=False, default=240,
                        help='Length of the simulation. [s]')
    parser.add_argument('--dt', type=float, required=False, default=0.01,
                        help='Length of the smallest step in simulation. [s]')
    parser.add_argument('--scale', type=int, required=False, default=150,
                        help='Scale of the simulation. Scale of 100 means that 1m is equal to 100px.')
    parser.add_argument('--speed', type=float, required=False, default=1,
                        help='Playback speed multiplier. High values may cause inaccuracy.')
    parser.add_argument('--path-depth', type=int, required=False, default=50,
                        help='Number of previous positions to track and show on path. 0 means infinite depth.')
    parser.add_argument('--ag', type=float, required=False, default=9.81,
                        help='Value of teh gravitational acceleration used in simulation. [m/s^2];')
    parser.add_argument('--mass', nargs=2, type=float, required=False, default=(1, 1),
                        help='Masses of bodies in pendulum. [kg] Enter in format: --mass M1 M2')
    parser.add_argument('--length', nargs=2, type=float, required=False, default=(1.1, 1),
                        help='Lengths of rods in pendulum. [m] Enter in format: --length L1 L2')
    parser.add_argument('--theta', nargs=2, type=float, required=False, default=(40, 160),
                        help='Initial angles of bodies in pendulum. [deg] Enter in format: --theta t1 t2')
    parser.add_argument('--dtheta', nargs=2, type=float, required=False, default=(0, 0),
                        help='Initial angular velocities of bodies in pendulum. [deg/s] '
                            'Enter in format: --dtheta dt1 dt2')

    # Parsing args and starting the simulation
    args = parser.parse_args()
    sim = Simulation(args.res, args.fps, args.time, args.dt, args.scale, args.speed, args.path_depth,
                     args.ag, args.mass, args.length, args.theta, args.dtheta)
    sim.run()
```

Na powyższym screenie widzimy w jaki sposób program pobiera argumenty z terminala. Na ich podstawie, po uruchomieniu tworzy się symulacja.

Widoczny poniżej fragment kodu jest odpowiedzialny za tworzenie głównego okna symulacji. Generuje także metodykę rysowania trajektorii ruchu wahadła oraz odświeżania kolejnych klatkek.

```python
import pygame
import os


class Display:
    # RGB tuples dictionary
    COLORS = {
        'BLACK': (0, 0, 0),
        'RED': (255, 0, 0),
        'WHITE': (255, 255, 255)
    }

    def __init__(self, res):
        # Setting resolution of the simulation
        self.width, self.height = res

        # Initializing pygame window
        self.window = pygame.display.set_mode((self.width, self.height))
        pygame.display.set_caption('Double pendulum simulation')
        pygame.display.flip()

        # Initializing pygame font
        pygame.font.init()
        font_path = os.path.join(os.path.dirname(__file__), 'resources/RobotoMono.ttf')
        self.font = pygame.font.Font(font_path, 18)

    def draw_rod(self, start, end):
        """ Draw a rod. """
        pygame.draw.line(self.window, self.COLORS['WHITE'], start, end, 4)

    def draw_body(self, pos):
        """ Draw a body. """
        pygame.draw.circle(self.window, self.COLORS['WHITE'], pos, 20)

    def draw_path(self, path, path_depth):
        """ Draw a path. """

        # Doesn't do anything if path contains too few elements
        if len(path) > 1:
            if path_depth == 0 or path_depth >= len(path):  # Infinite path or path too short to slice
                start = 1
            else:
                start = len(path) - path_depth

            for point in range(start, len(path)):
                pygame.draw.line(self.window, self.COLORS['RED'], path[point], path[point - 1], 2)

    def draw_info(self, info_str, pos):
        """ Draw a line of text at given position. """
        text = self.font.render(info_str, 1, self.COLORS['WHITE'])
        self.window.blit(text, pos)

    @staticmethod
    def handle_events():
        """ Handles events in window loop. """
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

    def next_frame(self, delay):
        """ Update a frame, and prepare surface for next one after given delay. """
        pygame.display.update()
        pygame.time.wait(delay)
        self.window.fill(self.COLORS['BLACK'])

```

```python
import numpy as np
from scipy.integrate import odeint


def deg_to_rad(theta_deg):
    """ Convert degrees to radians. """
    return theta_deg * 0.0174532925


def rad_to_deg(theta_rad):
    """ Convert radians to degrees. """
    return theta_rad * 57.2957795


class DoublePendulum:
    def __init__(self, mass, length, theta, d_theta):
        # Setting constants and initial variables
        self.mass_1, self.mass_2 = mass
        self.length_1, self.length_2 = length
        self.initial_theta_1_deg, self.initial_theta_2_deg = theta
        self.initial_d_theta_1_deg, self.initial_d_theta_2_deg = d_theta

        # Converting initial variables to radians for solving ODEs
        self.initial_theta_1 = deg_to_rad(self.initial_theta_1_deg)
        self.initial_theta_2 = deg_to_rad(self.initial_theta_2_deg)
        self.initial_d_theta_1 = deg_to_rad(self.initial_d_theta_1_deg)
        self.initial_d_theta_2 = deg_to_rad(self.initial_d_theta_2_deg)

        # Initializing result variables before solving ODEs
        self.theta_1, self.theta_2 = (None, None)
        self.d_theta_1, self.d_theta_2 = (None, None)
        self.x_1, self.y_1, self.x_2, self.y_2 = (None, None, None, None)

    def derive(self, y, t, ag, _):
        """ Differential equations used in the simulation.
        Source: https://www.math24.net/double-pendulum/"""

        theta_1, d_theta_1, theta_2, d_theta_2 = y
        sine, cosine = np.sin(theta_1 - theta_2), np.cos(theta_1 - theta_2)

        theta_1_dot, theta_2_dot = d_theta_1, d_theta_2

        d_theta_1_dot = ((self.mass_2 * ag * np.sin(theta_2) * cosine - self.mass_2 * sine *
                          (self.length_1 * d_theta_1 ** 2 * cosine + self.length_2 * d_theta_2 ** 2) -
                          (self.mass_1 + self.mass_2) * ag * np.sin(theta_1)) / self.length_1 /
                         (self.mass_1 + self.mass_2 * sine ** 2))

        d_theta_2_dot = (((self.mass_1 + self.mass_2) * (self.length_1 * d_theta_1 ** 2 * sine - ag * np.sin(theta_2) +
                          ag * np.sin(theta_1) * cosine) + self.mass_2 * self.length_2 * d_theta_2 ** 2 *
                         sine * cosine) / self.length_2 / (self.mass_1 + self.mass_2 * sine ** 2))

        return theta_1_dot, d_theta_1_dot, theta_2_dot, d_theta_2_dot

    def solve(self, time_range, ag):
        """ Solve ODEs in given time range, using given gravitational acceleration. """

        # Converting initial values to numpy array
        initial_conditions = np.array([self.initial_theta_1, self.initial_d_theta_1,
                                       self.initial_theta_2, self.initial_d_theta_2])

        # Solving ODEs
        result = odeint(self.derive, initial_conditions, time_range, args=(ag, None))
        self.theta_1, self.d_theta_1 = result[:, 0], result[:, 1]
        self.theta_2, self.d_theta_2 = result[:, 2], result[:, 3]

        self.convert_to_cartesian()

    def convert_to_cartesian(self):
        """ Convert solution to cartesian units. """
        self.x_1 = self.length_1 * np.sin(self.theta_1)
        self.y_1 = -self.length_1 * np.cos(self.theta_1)
        self.x_2 = self.x_1 + self.length_2 * np.sin(self.theta_2)
        self.y_2 = self.y_1 - self.length_2 * np.cos(self.theta_2)

    def get_positions(self, i):
        """ Get raw cartesian positions. """
        return self.x_1[i], self.y_1[i], self.x_2[i], self.y_2[i]

    def get_scaled_positions(self, i, scale):
        """ Get scaled for display cartesian positions using given scale factor. """
        scaled_x_1 = self.x_1[i] * scale
        scaled_y_1 = self.y_1[i] * scale
        scaled_x_2 = self.x_2[i] * scale
        scaled_y_2 = self.y_2[i] * scale

        return scaled_x_1, scaled_y_1, scaled_x_2, scaled_y_2

    def get_angles(self, i):
        """ Get angles in degrees. """
        return rad_to_deg(self.theta_1[i]), rad_to_deg(self.theta_2[i])

    def get_angular_velocities(self, i):
        """ Get angular velocities in degrees per second. """
        return rad_to_deg(self.d_theta_1[i]), rad_to_deg(self.d_theta_2[i])

```

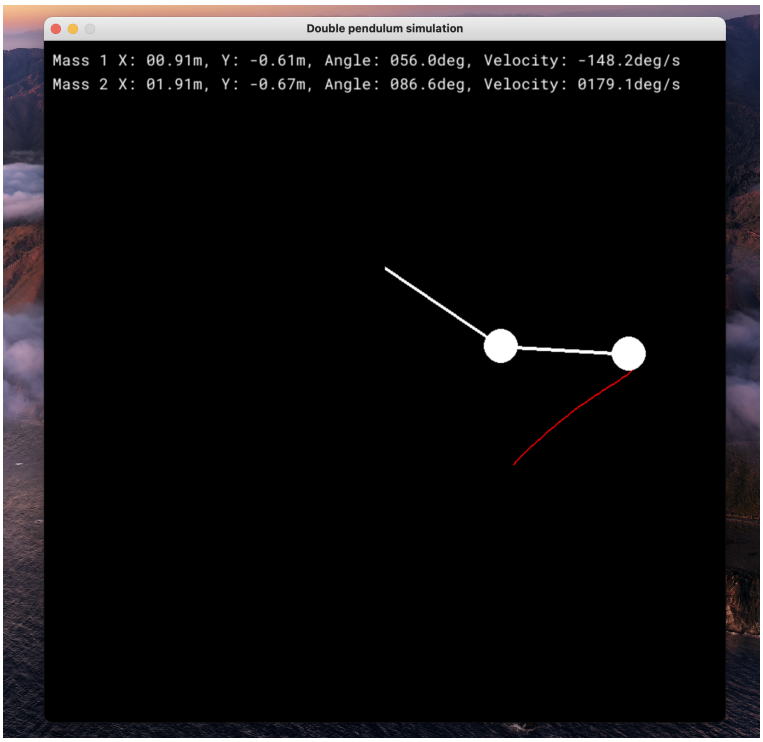
Powyższy kod przedstawia model wahadła wraz z metodami obliczeniowymi oraz dodatkowymi metodami dla jednostek. Pobiera on wartości początkowe i dostarcza metody zwracające dane w danej chwili czasu.

```python
import numpy as np
from display import Display
from double_pendulum import DoublePendulum


class Simulation:
    def __init__(self, res, fps, time, dt, scale, speed, path_depth, ag, mass, length, theta, d_theta):
        self.running = False  # Simulation not started yet at this point

        # Setting simulation variables
        self.width, self.height = res
        self.fps = fps
        self.time, self.dt = time, dt
        self.points_count = self.time / self.dt
        self.scale = scale
        self.speed = speed
        self.path_depth = path_depth
        self.path = []

        # Setting variables for solve method.
        self.time_range = np.arange(0, self.time + self.dt, self.dt)  # Range of time points
        self.ag = ag  # Gravitational acceleration

        # Creating display and pendulum objects; Solving equations
        self.pendulum = DoublePendulum(mass, length, theta, d_theta)
        self.pendulum.solve(self.time_range, self.ag)
        self.display = Display(res)

    def run(self):
        """ Simulation loop. """

        # Start the simulation
        self.running = True

        i = 0  # Iterable for simulation frames
        di = int((1 / self.fps / self.dt) * self.speed)  # Step size based on time density and frame rate

        # Loop
        while self.running:
            # Getting start point and all of the pendulum positions
            x_start, y_start = (int(self.width / 2), int(self.height / 3))
            x_1_scaled, y_1_scaled, x_2_scaled, y_2_scaled = self.pendulum.get_scaled_positions(i, self.scale)
            start_pos = (x_start, y_start)
            x_1_pos = (x_start + x_1_scaled, y_start - y_1_scaled)
            x_2_pos = (x_start + x_2_scaled, y_start - y_2_scaled)

            # Getting raw positions and velocities to show live info
            x_1_raw, y_1_raw, x_2_raw, y_2_raw = self.pendulum.get_positions(i)
            angle_1, angle_2 = self.pendulum.get_angles(i)
            velocity_1, velocity_2 = self.pendulum.get_angular_velocities(i)

            # Setting strings ready to be drawn on display
            info_str = 'Mass {} X: {:05.2f}m, Y: {:05.2f}m, Angle: {:05.1f}deg, Velocity: {:06.1f}deg/s'
            m_1_info_str = info_str.format(1, x_1_raw, y_1_raw, angle_1 % 360, velocity_1)
            m_2_info_str = info_str.format(2, x_2_raw, y_2_raw, angle_2 % 360, velocity_2)

            # Drawing rods
            self.display.draw_rod(start_pos, x_1_pos)
            self.display.draw_rod(x_1_pos, x_2_pos)

            self.display.draw_path(self.path, self.path_depth)  # Drawing path

            # Drawing bodies
            self.display.draw_body(x_1_pos)
            self.display.draw_body(x_2_pos)

            # Drawing live info
            self.display.draw_info(m_1_info_str, (10, 10))
            self.display.draw_info(m_2_info_str, (10, 38))

            # Handling events and transition into next iteration
            self.display.next_frame(int(1000 / self.fps))
            self.display.handle_events()

            # Inserting current second pendulum position into path
            self.path.append(x_2_pos)

            # Iterating and stopping the simulation at the end of solved time range
            i += di
            if i > self.points_count:
                self.running = False
```

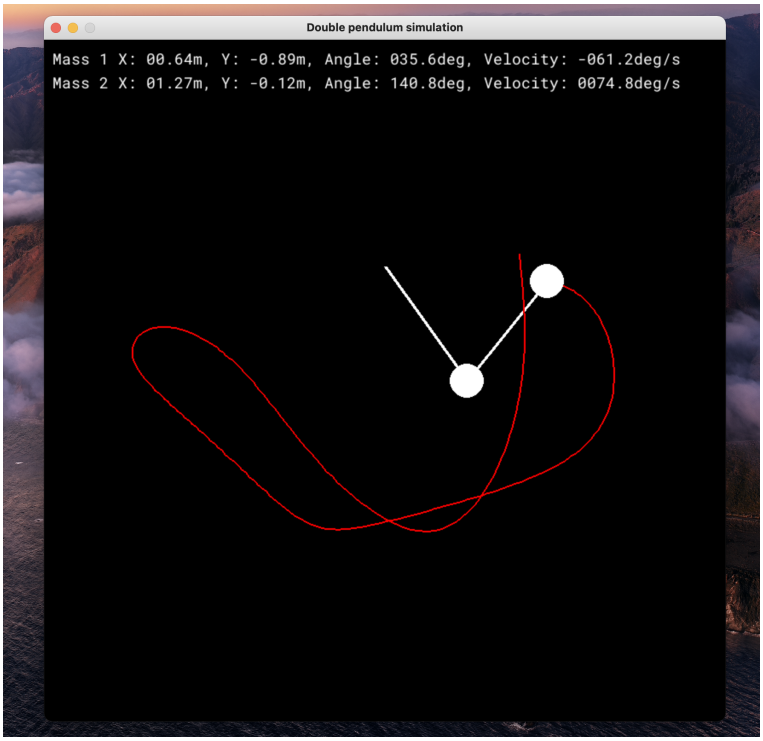
Fragment kodu umieszczony na poprzedniej stronie to controller. Służy on do tworzenia instance display oraz wahadła. W pętli wywołuje także pozostałe, przedstawione fragmenty kodu.

3 Wyniki

Uruchomiony program od strony użytkownika wygląda następująco. Wyświetla on dane dla naszego wahadła podwójnego oraz zaznacza trajektorię jego ruchu.



Poniższa ilustracja przedstawia symulację dla wartości długości ścieżki równej zero.



4 Podsumowanie

Wykorzystując potencjał języku programowania jakim jest Python oraz jego rozbudowane zaplecze bibliotek, udało nam się stworzyć opisywany w celu pracy projekt, mający na celu symulowanie ruchu wahadła podwójnego. Odtworzenie podobnego programu nie powinno stanowić problemu, bazując na zamieszczonych przez nas fragmentach kodu.

Zasada działania symulacji nie jest skomplikowana i nie wymaga szerokiego spektrum wiedzy z zakresu fizyki. Całość zakreślanej w wynikach trajektorii opiera się na teorii opisanej przez nas we wstępie, a więc dwóch wzorach matematycznych, służących do opisu ruchu wahadła podwójnego. Od strony użytkownika program prezentuje się czytelnie i wyświetla wszystkie niezbędne parametry.

5 Bibliografia

Zagadnienie wahadła podwójnego:

http://www.kms.polsl.pl/mi/pelne_29/04_29_60.pdf?fbclid=IwAR2Ny4qvFWv0_AMd2BPupc6PFmY1gVn2TMuI2SK1OLXtNP6Ima3ItJMtthY

Plik z zajęć (ODE2 12.10 - 12.18 *Chaotic Pendulum*)