Projekt Robot 2D

Stanisław Białecki, Andrzej Datta, Norbert Grzenkowicz

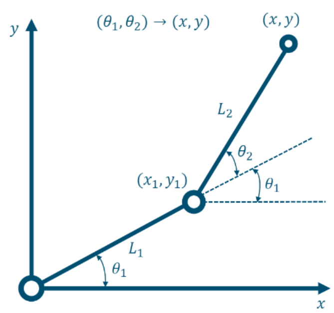
31 maja 2022

1 Cel ćwiczenia

Celem ćwiczenia było stworzenie symulacji działania robota 2D.

2 Obliczenia

2.1 Kinematyka prosta



Rysunek 1: Schemat kinematyki prostej

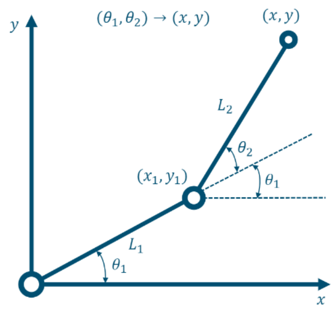
	a_i	α_i	d_i	ϕ
1	$a_1 = L_1$	0	0	d_1
2	$a_2 = L_2$	0	0	d_2

Tablica 1: Notacja D-H

Wzory potrzebne przy wyznaczaniu pozycji maniupulatora:

$$x_1 = L_1 \cos \phi_1$$
$$y_1 = L_1 \sin \phi_1$$
$$x = L_1 \cos \phi_1 + L_2 \cos \phi_1 + \phi_2$$
$$y = L_1 \sin \phi_1 + L_2 \sin \phi_1 + \phi_2$$

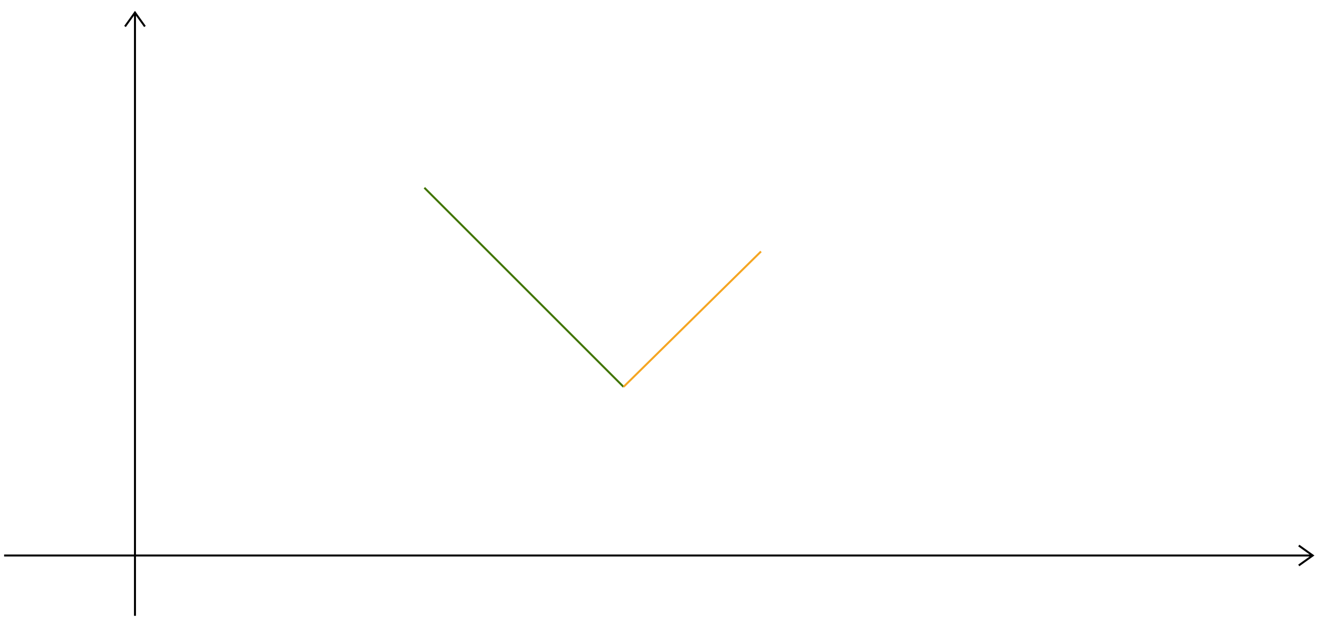
2.2 Kinematyka odwrotna



Rysunek 2: Schemat kinematyki odwrotnej

Potrzebne wzory

$$\phi_1 = \arctan \frac{y}{x} - \arccos \frac{L_1^2+L^2-L_2^2}{2L_1L_2} \ \phi_2 = \pi - \arccos \frac{L_1^2+L^2-L_2^2}{2L_1L_2}$$



2.3 Python

```python
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

class robot_model:
    def __init__(self,
                 init_state = [120, 0, 120, 0],
                 L1=1.0,  # dlugosc ramienia 1
                 L2=1.0,  # dlugosc ramienia 2
                 M1=1.0,  # masa ramienia 1 [kg]
                 M2=1.0,  # mass ramienia 2 [kg]
                 G=9.8,  # grawitacja m/s^2
                 origin=(0, 0)):
        self.init_state = np.asarray(init_state, dtype='float')
        self.params = (L1, L2, M1, M2, G)
        self.origin = origin
        self.time_elapsed = 0
        self.dt = 0.05
        self.state = self.init_state * np.pi / 180.
        self.trajectory = [[], [], []]

        self.__theta_target = []

        self.p = np.zeros(2)
        self.__p_target = None
        self.__p_start = None

        # Parametry macierzy DH
        theta_0 = [0.0,0.0]
        a       = [L1, L2]
        d       = [0.0, 0.0]
        alpha   = [0.0, 0.0]
        self.dh_params = {"theta": theta_0, "a": a, "d": d, "alpha": alpha}

        # Dlugosc ramienia [m]
        self.L  = [L1, L2]
        # Polowa dlugosci ramienia [m]
        self.lg = [L1/2, L2/2]
        # Masa [kg]
        self.m  = [M1, M2]
        # Moment inercyjny [kg.m^2]
        self.I  = [(1/3)*(M1)*(L1**2), (1/3)*(M2)*(L2**2)]
        # Przyspieszenie grawitacyjne  [m/s^2]
        self.g  = G
        # Czas animacji
        self.t = np.arange(0.0, 20, self.dt)

        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111, autoscale_on = False, xlim = (-2, 2), ylim
 = (-2, 2))
        self.ax.grid()
        line, = self.ax.plot([], [], 'o-', lw = 2)
        __line1, = self.ax.plot([], [], 'o-', lw = 2)
        __line2, = self.ax.plot([], [], 'o-', lw = 2)
        __line3,  = self.ax.plot([], [], 'o-', lw = 2)
        __line4,  = self.ax.plot([], [], 'o-', lw = 2)

        self.__line = [__line1, __line2, __line3, __line4]
        self.__animation_dMat = np.zeros((1, 4), dtype=np.float64)


    def position(self):
        (L1, L2, M1, M2, G) = self.params

        x = np.cumsum([self.origin[0],
                       L1 * sin(self.state[0]),
                       L2 * sin(self.state[2])])
```

```python
        y = np.cumsum([self.origin[1],
                       -L1 * cos(self.state[0]),
                       -L2 * cos(self.state[2])])
        return x.tolist(), y.tolist()

    def dstate_dt(self, input_p, t):
        theta_1  = input_p[0]; theta_2  = input_p[2]
        dtheta_1 = input_p[1]; dtheta_2 = input_p[3]

        M_Mat    = np.matrix([
            [self.I[0] + self.I[1] + self.m[0] * (self.lg[0]**2) + self.m[1] * ((self.L[0]**2) + (self.lg[1]**2) + 2 * self.L[0] * self.lg[1] * np.cos(theta_2)), self.I[1] + self.m[1] * ((self.lg[1]**2) + self.L[0] * self.lg[1] * np.cos(theta_2))],
            [self.I[1] + self.m[1] * ((self.lg[1]**2) + self.L[0] * self.lg[1] * np.cos(theta_2)), self.I[1] + self.m[1] * (self.lg[1]**2)]
        ])

        b_Mat    = np.matrix([
            [(-1) * self.m[1] * self.L[0] * self.lg[1] * dtheta_2 * (2 * dtheta_1 + dtheta_2) * np.sin(theta_2)],
            [self.m[1] * self.L[0] * self.lg[1] * (dtheta_1**2) *np.sin(theta_2)]
        ])

        g_Mat    = np.matrix([
            [self.m[0] * self.g * self.lg[0] * np.cos(theta_1) + self.m[1] * self.g * (self.L[0] * np.cos(theta_1) + self.lg[1] * np.cos(theta_1 + theta_2))],
            [self.m[1] * self.g * self.lg[1] * np.cos(theta_1 + theta_2)]
        ])

        # Ordinary Differential Equations (ODE)
        ode_r = np.linalg.inv(M_Mat).dot(-b_Mat - g_Mat)

        return [dtheta_1, ode_r[0][0], dtheta_2, ode_r[1][0]]

    def forward_kin(self):

        self.p[0] = round(self.dh_params["a"][0]*np.cos(self.dh_params["theta"][0]) + self.dh_params["a"][1]*np.cos(self.dh_params["theta"][0] + self.dh_params["theta"][1]), 5)
        self.p[1] = round(self.dh_params["a"][0]*np.sin(self.dh_params["theta"][0]) + self.dh_params["a"][1]*np.sin(self.dh_params["theta"][0] + self.dh_params["theta"][1]), 5)

    def forward_kinematics(self, theta):
        self.__theta_target = np.zeros(2)
        self.__theta_target[0] = theta[0]
        self.__theta_target[1] = theta[1]

        self.dh_params["theta"] = self.__theta_target

        self.forward_kin()


    def inverse_kinematics(self, p):

        theta_aux      = np.zeros(2)
        self.__p_target = np.zeros(2)
        self.__p_target[0] = p[0]
        self.__p_target[1] = p[1]

        cosT_beta_numerator   = ((self.dh_params["a"][0]**2) + (self.__p_target[0]**2 + self.__p_target[1]**2) - (self.dh_params["a"][1]**2))
        cosT_beta_denumerator = (2*self.dh_params["a"][0]*np.sqrt(self.__p_target[0]**2 + self.__p_target[1]**2))

        # THETA 1
        if cosT_beta_numerator/cosT_beta_denumerator > 1:
            theta_aux[0] = np.arctan2(self.__p_target[1], self.__p_target[0])
            print('[INFO] Theta 1 Error: ', self.__p_target[0], self.__p_target[1])
        elif cosT_beta_numerator/cosT_beta_denumerator < -1:
            theta_aux[0] = np.arctan2(self.__p_target[1], self.__p_target[0]) - np.pi
```

```python
            print('[INFO] Theta 1 Error: ', self.__p_target[0], self.__p_target[1])
        else:
            theta_aux[0] = np.arctan2(self.__p_target[1], self.__p_target[0]) + np.arccos(cosT_beta_numerator/cosT_beta_denumerator)

        cosT_alpha_numerator   = (self.dh_params["a"][0]**2) + (self.dh_params["a"][1]**2) - (self.__p_target[0]**2 + self.__p_target[1]**2)
        cosT_alpha_denumerator = (2*(self.dh_params["a"][0]*self.dh_params["a"][1]))

        # THETA2
        if cosT_alpha_numerator/cosT_alpha_denumerator > 1:
            theta_aux[1] = np.pi
            print('[INFO] Theta 2 Error: ', self.__p_target[0], self.__p_target[1])
        elif cosT_alpha_numerator/cosT_alpha_denumerator < -1:
            theta_aux[1] = 0.0
            print('[INFO] Theta 2 Error: ', self.__p_target[0], self.__p_target[1])
        else:
            theta_aux[1] = np.arccos(cosT_alpha_numerator/cosT_alpha_denumerator) - np.pi

        self.theta = theta_aux
        self.forward_kinematics(self.theta)

    def generate_trajectory(self, trajectory_point):

        self.__p_target = trajectory_point[0], trajectory_point[1]

        start_theta  = self.dh_params["theta"]

        self.inverse_kinematics(self.__p_target)
        self.__theta_target = self.theta
        x   = []
        y   = []

        self.inverse_kinematics(trajectory_point)

        # self.step()

        self.inverse_kinematics(trajectory_point)
        target_theta = self.__theta_target

        start_theta_dt  = np.linspace(start_theta[0], target_theta[0], 100)
        target_theta_dt = np.linspace(start_theta[1], target_theta[1], 100)

        # start_theta_dt  = self.state[0]
        # target_theta_dt = self.state[1]

        for i in range(len(start_theta_dt)):
            self.forward_kinematics([start_theta_dt[i], target_theta_dt[i]])
            x.append(self.p[0])
            y.append(self.p[1])

        # x, y = self.position()
        self.trajectory[0] = x
        self.trajectory[1] = y

    def step(self):
        self.state = integrate.odeint(self.dstate_dt, self.state, np.linspace(0, 128, 128))
        self.time_elapsed += self.dt

    def __animation_data_generation(self):
        self.__animation_dMat = np.zeros((len(self.trajectory[0]), 4), dtype=np.float64)

        for i in range(len(self.trajectory[0])):
            self.inverse_kinematics([self.trajectory[0][i], self.trajectory[1][i]])
            self.__animation_dMat[i][0] = self.dh_params["a"][0]*np.cos(self.dh_params["theta"][0])
            self.__animation_dMat[i][1] = self.dh_params["a"][0]*np.sin(self.dh_params["theta"][0])
```

```
            self.__animation_dMat[i][2] = self.p[0]
            self.__animation_dMat[i][3] = self.p[1]

    def init_animation(self):
        self.__animation_data_generation()

        self.__line[0].set_data([0.0, self.__animation_dMat[0][0]], [0.0, self.__animation_dMat[0][1]])
        self.__line[1].set_data([self.__animation_dMat[0][0], self.__animation_dMat[0][2]], [self.__animation_dMat[0][1], self.__animation_dMat[0][3]])
        self.__line[2].set_data(self.__animation_dMat[0][0], self.__animation_dMat[0][1])
        self.__line[3].set_data(self.__animation_dMat[0][2], self.__animation_dMat[0][3])

        return [self.__line[0], self.__line[1], self.__line[2], self.__line[3]]

    def start_animation(self, i):

        self.__line[0].set_data([0.0, self.__animation_dMat[i][0]], [0.0, self.__animation_dMat[i][1]])
        self.__line[1].set_data([self.__animation_dMat[i][0], self.__animation_dMat[i][2]], [self.__animation_dMat[i][1], self.__animation_dMat[i][3]])
        self.__line[2].set_data(self.__animation_dMat[i][0], self.__animation_dMat[i][1])
        self.__line[3].set_data(self.__animation_dMat[i][2], self.__animation_dMat[i][3])

        return [self.__line[0], self.__line[1], self.__line[2], self.__line[3]]

def animate_animation():
    p1, p2 = input("WPISZ PORZADANY PUNKT X (OD 0 DO 2): "), input("WPISZ PORZADANY PUNKT Y (OD 0 DO 2): ")

    robot2D = robot_model([240.0, 0.0, 240.0, 0.0])
    dt = 1./30 # 30 fps

    robot2D.generate_trajectory([p1, p2]) #([-0.5, 0.5])

    line, = robot2D.ax.plot([], [], 'o-', lw=2, animated=True)
    time_text = robot2D.ax.text(0.02, 0.95, '', transform=robot2D.ax.transAxes)
    energy_text = robot2D.ax.text(0.02, 0.90, '', transform=robot2D.ax.transAxes)

    animator = animation.FuncAnimation(robot2D.fig, robot2D.start_animation, init_func=robot2D.init_animation, frames=len(robot2D.trajectory[0]), interval=2)

    plt.show()

animate_animation()
```

3 Podsumowanie

Zadanie zostało zrealizowane zgodnie z wymaganiami. Symulacja robota 2d odbywa sie w czasie ciagłym.

Literatura

[1] Dr inż. Krzysztof Armiński, *Tematy*