```
"""
    @brief: This file contains the UR5 class, which is used to control the UR5 robot in Webots
    @version: v1.0
    @date: 2023/6/9
    @author: Allan Souza Almeida

    @how_to_use:
        1. Import the class: from ur5 import UR5
        2. Create an instance of the class: ur5 = UR5()
        3. Use the functions to control the robot angles (FK): ur5.move_to_config([0, 0, 0, 0, 0, 0])
        4. Use the functions to control the robot poses (IK): ur5.move_to_pose([0.2, 0, 0.4], [pi/2, 0, 0])
        5. Use the functions to control the gripper: ur5.actuate_gripper(1)

"""

from skimage.transform import resize
import numpy as np
from math import pi, cos, sin
import math
from controller import Supervisor
from functools import reduce
from scipy.spatial.transform import Rotation
from keras.models import load_model
import matplotlib.pyplot as plt
from keras.applications.vgg16 import preprocess_input
PI = pi


def rot_z(theta):
    """
    Returns the rotation matrix around the z axis

    Parameters:
        theta (float): angle in radians

    Returns:
        R (np.array): rotation matrix
    """
    return np.array(
        [
            [np.cos(theta), -np.sin(theta), 0, 0],
            [np.sin(theta), np.cos(theta), 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, 1],
        ]
    )


def rot_x(theta):
    """
    Returns the rotation matrix around the x axis

    Parameters:
        theta (float): angle in radians

    Returns:
        R (np.array): rotation matrix
    """
    return np.array(
        [
            [1, 0, 0, 0],
            [0, np.cos(theta), -np.sin(theta), 0],
            [0, np.sin(theta), np.cos(theta), 0],
            [0, 0, 0, 1],
        ]
    )


def rot_y(theta):
    """
    Returns the rotation matrix around the y axis

    Parameters:
        theta (float): angle in radians

    Returns:
        R (np.array): rotation matrix
    """
    return np.array(
        [
            [np.cos(theta), 0, np.sin(theta), 0],
```

```python
            [0, 1, 0, 0],
            [-np.sin(theta), 0, np.cos(theta), 0],
            [0, 0, 0, 1],
        ]
    )


def limit_angle(angle):
    """
    Limits the angle between -pi and pi

    Parameters:
        angle (float): angle in radians

    Returns:
        angle (float): angle in radians between -pi and pi
    """
    angle_mod = angle % (2 * np.pi)
    if angle_mod > np.pi:
        return angle_mod - 2 * np.pi
    else:
        return angle_mod


def build_matrix(pos: "np.ndarray", rot: "np.ndarray", euler: "str" = "XYZ"):
    """
    Builds the transformation matrix from position and Euler angles

    Parameters:
        pos (list[float | int]): position
        rot (list[float | int]): XYZ rotation (Euler angles)
        euler (str): Euler angle order (XYZ or ZYX)

    Returns:
        R (np.array): transformation matrix
    """
    Rx = rot_x(rot[0])
    Ry = rot_y(rot[1])
    Rz = rot_z(rot[2])
    if euler == "XYZ":
        R = reduce(np.dot, [Rx, Ry, Rz])
    elif euler == "ZYX":
        R = reduce(np.dot, [Rz, Ry, Rx])
    R[0][3] = pos[0]
    R[1][3] = pos[1]
    R[2][3] = pos[2]
    return R


def matrix_error(T1: "np.ndarray", T2: "np.ndarray"):
    """
    Calculates the error between two transformation matrices

    Parameters:
        T1 (np.ndarray): transformation matrix
        T2 (np.ndarray): transformation matrix

    Returns:
        angle_error (float): angle error between the two matrices
        pos_error (float): position error between the two matrices
    """
    R1 = T1[:3, :3]
    R2 = T2[:3, :3]
    R_diff = np.dot(R1, R2.T)
    r = Rotation.from_matrix(R_diff)
    axis = r.as_rotvec()
    angle = np.linalg.norm(axis)*180/PI
    P1 = T1[:3, 3]
    P2 = T2[:3, 3]
    dist = np.linalg.norm(P1 - P2)*1000
    return angle, dist


def forward_kinematics(theta: "list[float | int] | np.ndarray"):
    """
    Defines Denavit-Hartenberger parameters for UR5 and calculates
    forward kinematics

    Parameters:
        theta (list[float | int]): joint angles in radians

    Returns:
        T (tuple[np.ndarray, np.ndarray]): total transformation matrix and
```

```
        transformation matrices for each joint
    """
    d1 = 0.1625
    a2 = 0.425
    a3 = 0.3922
    d4 = 0.1333
    d5 = 0.0997
    d6 = 0.0996+0.1237
    dh_table = np.array(
        [
            [0, PI / 2, d1, 0],
            [a2, 0, 0, PI / 2],
            [a3, 0, 0, 0],
            [0, -PI / 2, d4, -PI / 2],
            [0, PI / 2, d5, 0],
            [0, 0, d6, 0],
        ]
    )

    A = np.array(
        [
            np.array(
                [
                    [
                        cos(theta[i] + dh_table[i][3]),
                        -sin(theta[i] + dh_table[i][3]) * cos(dh_table[i][1]),
                        sin(theta[i] + dh_table[i][3]) * sin(dh_table[i][1]),
                        dh_table[i][0] * cos(theta[i] + dh_table[i][3]),
                    ],
                    [
                        sin(theta[i] + dh_table[i][3]),
                        cos(theta[i] + dh_table[i][3]) * cos(dh_table[i][1]),
                        -cos(theta[i] + dh_table[i][3]) * sin(dh_table[i][1]),
                        dh_table[i][0] * sin(theta[i] + dh_table[i][3]),
                    ],
                    [0, sin(dh_table[i][1]), cos(
                        dh_table[i][1]), dh_table[i][2]],
                    [0, 0, 0, 1],
                ]
            )
            for i in range(6)
        ]
    )

    T = reduce(np.dot, A)
    return T, A


def transform(theta: "int | float", idx):
    """
    Calculate the transformation matrix between two consecutive frames

    Ex: T_0_1, T_1_2, T_2_3, T_3_4, T_4_5, T_5_6

    Parameters:
        theta (float | int): joint angle in radians
        idx (int): index of the transformation matrix

    Returns:
        T (np.array): transformation matrix
    """
    d1 = 0.1625
    a2 = 0.425
    a3 = 0.3922
    d4 = 0.1333
    d5 = 0.0997
    d6 = 0.0996+0.1237
    dh_table = np.array(
        [
            [0, PI / 2, d1, 0],
            [a2, 0, 0, PI / 2],
            [a3, 0, 0, 0],
            [0, -PI / 2, d4, -PI / 2],
            [0, PI / 2, d5, 0],
            [0, 0, d6, 0],
        ]
    )

    th = np.array(
        [
            [
                cos(theta + dh_table[idx][3]),
                -sin(theta + dh_table[idx][3]) * cos(dh_table[idx][1])
```

```
                -sin(theta + dh_table[idx][3]) * cos(dh_table[idx][1]),
                sin(theta + dh_table[idx][3]) * sin(dh_table[idx][1]),
                dh_table[idx][0] * cos(theta + dh_table[idx][3]),
            ],
            [
                sin(theta + dh_table[idx][3]),
                cos(theta + dh_table[idx][3]) * cos(dh_table[idx][1]),
                -cos(theta + dh_table[idx][3]) * sin(dh_table[idx][1]),
                dh_table[idx][0] * sin(theta + dh_table[idx][3]),
            ],
            [0, sin(dh_table[idx][1]), cos(
                dh_table[idx][1]), dh_table[idx][2]],
            [0, 0, 0, 1],
        ]
    )
    return th


def inverse_kinematics(th: "np.ndarray", shoulder="left", wrist="down", elbow="up"):
    """
    Calculates inverse kinematics for UR5

    Parameters:
        th (np.ndarray): transformation matrix
        shoulder (str): 'left' or 'right'
        wrist (str): 'up' or 'down'
        elbow (str): 'up' or 'down'

    Returns:
        theta (list[float]): joint angles in radians
    """
    try:
        a2 = 0.425
        a3 = 0.3922
        d4 = 0.1333
        d6 = 0.0996+0.1237
        o5 = th.dot(np.array([[0, 0, -d6, 1]]).T)
        xc, yc, zc = o5[0][0], o5[1][0], o5[2][0]

        # Theta 1
        psi = math.atan2(yc, xc)
        phi = math.acos(d4 / np.sqrt(xc**2 + yc**2))
        theta1 = np.array([psi - phi + PI / 2, psi + phi + PI / 2])
        T1 = np.array([limit_angle(theta1[0]), limit_angle(theta1[1])])
        if shoulder == "left":
            theta1 = T1[0]
        else:
            theta1 = T1[1]

        # Theta 5
        P60 = np.dot(th, np.array([[0, 0, 0, 1]]).T)
        x60 = P60[0][0]
        y60 = P60[1][0]
        z61 = x60 * np.sin(T1) - y60 * np.cos(T1)
        T5 = np.array([np.arccos((z61 - d4) / d6), -
                       np.arccos((z61 - d4) / d6)]).T
        if shoulder == "left":
            T5 = T5[0]
            if wrist == "up":
                theta5 = T5[0]
            else:
                theta5 = T5[1]
        else:
            T5 = T5[1]
            if wrist == "down":
                theta5 = T5[0]
            else:
                theta5 = T5[1]

        # Theta 6
        th10 = transform(theta1, 0)
        th01 = np.linalg.inv(th10)
        th16 = np.linalg.inv(np.dot(th01, th))
        z16_y = th16[1][2]
        z16_x = th16[0][2]
        theta6 = math.atan2(-z16_y / np.sin(theta5),
                            z16_x / np.sin(theta5)) + PI
        theta6 = limit_angle(theta6)

        # Theta 3
        th61 = np.dot(th01, th)
        th54 = transform(theta5, 4)
        th65 = transform(theta6, 5)
```

```
        inv = np.linalg.inv(np.dot(th54, th65))
        th41 = np.dot(th61, inv)
        p31 = np.dot(th41, np.array([[0, d4, 0, 1]]).T) - \
            np.array([[0, 0, 0, 1]]).T

        p31_x = p31[0][0]
        p31_y = p31[1][0]
        D = (p31_x**2 + p31_y**2 - a2**2 - a3**2) / (2 * a2 * a3)
        T3 = np.array(
            [math.atan2(-np.sqrt(1 - D**2), D),
             math.atan2(np.sqrt(1 - D**2), D)]
        )
        if shoulder == "left":
            if elbow == "up":
                theta3 = T3[0]
            else:
                theta3 = T3[1]
        else:
            if elbow == "up":
                theta3 = T3[1]
            else:
                theta3 = T3[0]

        # Theta 2
        delta = math.atan2(p31_x, p31_y)
        epsilon = math.acos(
            (a2**2 + p31_x**2 + p31_y**2 - a3**2)
            / (2 * a2 * np.sqrt(p31_x**2 + p31_y**2))
        )
        T2 = np.array([-delta + epsilon, -delta - epsilon])
        if shoulder == "left":
            theta2 = T2[0]
        else:
            theta2 = T2[1]

        # Theta 4
        th21 = transform(theta2, 1)
        th32 = transform(theta3, 2)
        inv = np.linalg.inv(np.dot(th21, th32))
        th43 = np.dot(inv, th41)
        x43_x = th43[0][0]
        x43_y = th43[1][0]
        theta4 = math.atan2(x43_x, -x43_y)

        return [theta1, theta2, theta3, theta4, theta5, theta6]
    except ValueError:
        raise ValueError("Posição inalcançável para o braço robótico")


class UR5:
    """
    This class defines the UR5 object and its functions
    """

    def __init__(self):
        """
        This function initializes the UR5 object and the simulation
        """
        print("Initializing the UR5 class...")
        self.supervisor = Supervisor()
        self.supervisor.simulationReset()
        self.timestep = int(self.supervisor.getBasicTimeStep())
        self.supervisor.step(self.timestep)
        self.joints = None
        self.camera = None
        self.bottle = None
        self.finger_joints = None
        self.finger_joint_limits = None
        print("Initializing the computer vision model...")
        self.model = load_model("computer_vision/vgg16.h5")
        self.init_handles()
        print("Ready!")

    def setup_control_mode(self):
        """
        This function sets up the control mode of the joints
        (velocity for the robot and position for the fingers)
        """
        for i, dev in enumerate(self.joints):
            dev.setPosition(float("inf"))
            dev.getPositionSensor().enable(self.timestep)

        for dev in self.finger_joints:
```

```
        dev.setVelocity(float(100))
        dev.getPositionSensor().enable(self.timestep)

def init_handles(self):
    """
    This function initiates the nodes
    """
    print("Initializing the nodes and handles...")
    self.camera = self.supervisor.getDevice("camera")
    self.bottle = self.supervisor.getFromDef("bottle")
    self.joints = [
        "shoulder_pan_joint",
        "shoulder_lift_joint",
        "elbow_joint",
        "wrist_1_joint",
        "wrist_2_joint",
        "wrist_3_joint",
    ]
    self.joint_sensors = [
        "shoulder_pan_joint_sensor",
        "shoulder_lift_joint_sensor",
        "elbow_joint_sensor",
        "wrist_1_joint_sensor",
        "wrist_2_joint_sensor",
        "wrist_3_joint",
    ]
    self.finger_joints = [
        "finger_1_joint_1",
        "finger_1_joint_2",
        "finger_1_joint_3",
        "finger_2_joint_1",
        "finger_2_joint_2",
        "finger_2_joint_3",
        "finger_middle_joint_1",
        "finger_middle_joint_2",
        "finger_middle_joint_3",
    ]
    self.finger_joint_sensors = [
        "finger_1_joint_1_sensor",
        "finger_1_joint_2_sensor",
        "finger_1_joint_3_sensor",
        "finger_2_joint_1_sensor",
        "finger_2_joint_2_sensor",
        "finger_2_joint_3_sensor",
        "finger_middle_joint_1_sensor",
        "finger_middle_joint_2_sensor",
        "finger_middle_joint_3_sensor",
    ]
    self.joints = [self.supervisor.getDevice(
        joint) for joint in self.joints]
    self.finger_joints = [
        self.supervisor.getDevice(joint) for joint in self.finger_joints
    ]
    self.joint_sensors = [
        self.supervisor.getDevice(sensor) for sensor in self.joint_sensors
    ]
    self.finger_joint_sensors = [
        self.supervisor.getDevice(s) for s in self.finger_joint_sensors
    ]
    self.finger_joint_limits = [
        [0.0695, 0.8],
        [0.01, 1],
        [-0.8, -0.0723],
        [0.0695, 0.8],
        [0.01, 1],
        [-0.8, -0.0723],
        [0.0695, 0.8],
        [0.01, 1],
        [-0.8, -0.0723],
    ]
    self.setup_control_mode()

def setup_camera(self):
    """
    This function enables the camera
    """
    self.camera.enable(self.timestep)
    self.supervisor.step(self.timestep)
    self.supervisor.step(self.timestep)

def get_image(self):
    """
    This function gets the image from the camera
```

```
        This function gets the image from the camera

        Returns:
            image (np.ndarray): numpy array of the image
        """
        img = self.camera.getImageArray()
        image = np.array(img)
        image = image.astype(np.uint8)
        image = image.reshape((512, 512, 3))
        return image

    def get_joint_angles(self):
        """
        This function gets the joint angles of the robot

        Returns:
            angles (np.ndarray): numpy array of the joint angles
        """
        angles = [joint.getPositionSensor().getValue()
                  for joint in self.joints]
        # angles[0] -= pi
        # angles[1] += pi / 2
        # angles[3] += pi / 2
        # angles[5] -= pi / 2
        return np.array(angles)

    def get_finger_angles(self):
        """
        This function gets the joint angles of the fingers

        Returns:
            angles (np.ndarray): numpy array of the joint angles
        """
        return np.array(
            [joint.getPositionSensor().getValue()
             for joint in self.finger_joints]
        )

    def get_ground_truth(self):
        """
        This function gets the ground truth of frame 6 relative to frame 0

        Returns:
            th0_6 (np.ndarray): Homogeneous transformation of frame 6 relative to frame 0
        """
        R6_world = np.array(
            self.supervisor.getFromDef("frame6").getOrientation()
        ).reshape(3, 3)
        T6_world = np.array(self.supervisor.getFromDef("frame6").getPosition()).reshape(
            3, 1
        )
        th6_world = np.hstack(
            (np.vstack((R6_world, np.zeros((1, 3)))), np.vstack((T6_world, 1)))
        )
        R0_world = np.array(
            self.supervisor.getSelf().getOrientation()).reshape(3, 3)
        T0_world = np.array(
            self.supervisor.getSelf().getPosition()).reshape(3, 1)
        th0_world = np.hstack(
            (np.vstack((R0_world, np.zeros((1, 3)))), np.vstack((T0_world, 1)))
        )
        thworld_0 = np.linalg.inv(th0_world)
        th6_0 = np.dot(thworld_0, th6_world)
        return th6_0

    def get_jacobian(self, velocities: np.ndarray = None):
        """
        Calculate Jacobian and get the end-effector velocities (linear and angular)
        from the joint velocities

        Parameters:
            velocities (np.ndarray): do not assign any values when using as standalone function

        Returns:
            qsi (np.ndarray): 6x1 vector containing 3 linear velocities (x, y, z) and
            3 angular velocities (x, y, z)
        """
        angles = self.get_joint_angles()
        if velocities is None:
            velocities = np.array(
                [j.getVelocity() for j in self.joints]
            ).reshape((6, 1))
        _, A = forward_kinematics(angles)
```

```python
    A10 = A[0]
    A20 = np.dot(A[0], A[1])
    A30 = np.dot(A20, A[2])
    A40 = np.dot(A30, A[3])
    A50 = np.dot(A40, A[4])
    A60 = np.dot(A50, A[5])
    Z0 = np.array([[0, 0, 1]]).T
    Z1 = A10[:3, 2].reshape(3, 1)
    Z2 = A20[:3, 2].reshape(3, 1)
    Z3 = A30[:3, 2].reshape(3, 1)
    Z4 = A40[:3, 2].reshape(3, 1)
    Z5 = A50[:3, 2].reshape(3, 1)
    O0 = np.zeros((3, 1))
    O1 = A10[:3, 3].reshape(3, 1)
    O2 = A20[:3, 3].reshape(3, 1)
    O3 = A30[:3, 3].reshape(3, 1)
    O4 = A40[:3, 3].reshape(3, 1)
    O5 = A50[:3, 3].reshape(3, 1)
    O6 = A60[:3, 3].reshape(3, 1)
    Jw = np.hstack((Z0, Z1, Z2, Z3, Z4, Z5))
    Jv = np.hstack(
        (
            np.cross(Z0.T, (O6 - O0).T).T,
            np.cross(Z1.T, (O6 - O1).T).T,
            np.cross(Z2.T, (O6 - O2).T).T,
            np.cross(Z3.T, (O6 - O3).T).T,
            np.cross(Z4.T, (O6 - O4).T).T,
            np.cross(Z5.T, (O6 - O5).T).T,
        )
    )
    J = np.vstack((Jv, Jw))
    qsi = np.dot(J, velocities)
    return qsi

def move_to_config(
    self, target: "list[float | int]", duration=None, graph=False, jacob=False
):
    """
    Move to configuration using quintic trajectory

    Args:
        target: list of target angles

        duration: time to reach target in seconds

        graph: whether to plot the trajectory

        jacob: wheter to calculate and return jacobian

    Returns:
        duration: time to reach target in seconds

        max_error: maximum final joint error in degrees

        mean_error: mean final joint error in degrees

        graphs: list of graphs if graph=True

        jacob: linear and angular end-effector velocities
    """
    self.supervisor.step(self.timestep)
    t0 = self.supervisor.getTime()
    v0 = np.zeros(6)
    vf = np.zeros(6)
    q0 = self.get_joint_angles()
    qf = np.array(target)
    a0 = np.zeros(6)
    af = np.zeros(6)
    if duration is None:
        duration = np.max(np.abs(qf - q0)) * (4 / (0.5 * PI))
        if duration < 1.5:
            duration = 1.5
    tf = t0 + duration
    A = np.array(
        [
            [1, t0, t0**2, t0**3, t0**4, t0**5],
            [0, 1, 2 * t0, 3 * t0**2, 4 * t0**3, 5 * t0**4],
            [0, 0, 2, 6 * t0, 12 * t0**2, 20 * t0**3],
            [1, tf, tf**2, tf**3, tf**4, tf**5],
            [0, 1, 2 * tf, 3 * tf**2, 4 * tf**3, 5 * tf**4],
            [0, 0, 2, 6 * tf, 12 * tf**2, 20 * tf**3],
        ]
    )
```

```
        b = np.array([q0, v0, a0, qf, vf, af])
        x = [np.linalg.solve(A, b[:, i]) for i in range(6)]
        time0 = self.supervisor.getTime()
        iterations = 0
        end_effector_vel = []
        vel_jacob = [[], [], [], [], [], []]
        pos = [[], [], [], [], [], []]
        vel = [[], [], [], [], [], []]
        acc = [[], [], [], [], [], []]
        jerk = [[], [], [], [], [], []]
        time_arr = [[], [], [], [], [], []]
        self.setup_control_mode()
        while self.supervisor.getTime() <= tf:
            t = self.supervisor.getTime()
            for idx, joint in enumerate(self.joints):
                joint.setVelocity(
                    x[idx][1]
                    + 2 * x[idx][2] * t
                    + 3 * x[idx][3] * t**2
                    + 4 * x[idx][4] * t**3
                    + 5 * x[idx][5] * t**4
                )
                if graph:
                    p = (
                        x[idx][0]
                        + x[idx][1] * t
                        + x[idx][2] * t**2
                        + x[idx][3] * t**3
                        + x[idx][4] * t**4
                        + x[idx][5] * t**5
                    )
                    v = (
                        x[idx][1]
                        + 2 * x[idx][2] * t
                        + 3 * x[idx][3] * t**2
                        + 4 * x[idx][4] * t**3
                        + 5 * x[idx][5] * t**4
                    )
                    a = (
                        2 * x[idx][2]
                        + 6 * x[idx][3] * t
                        + 12 * x[idx][4] * t**2
                        + 20 * x[idx][5] * t**3
                    )
                    j = 6 * x[idx][3] + 24 * x[idx][4] * \
                        t + 60 * x[idx][5] * t**2
                    time_arr[idx].append(t - time0)
                    pos[idx].append(p)
                    vel[idx].append(v)
                    acc[idx].append(a)
                    jerk[idx].append(j)
                if jacob:
                    v = (
                        x[idx][1]
                        + 2 * x[idx][2] * t
                        + 3 * x[idx][3] * t**2
                        + 4 * x[idx][4] * t**3
                        + 5 * x[idx][5] * t**4
                    )
                    vel_jacob[idx].append(v)
            if jacob:
                end_effector_vel.append(
                    self.get_jacobian(
                        velocities=np.array([vj[-1] for vj in vel_jacob]).reshape(
                            (6, 1)
                        )
                    )
                )
            self.supervisor.step(self.timestep)
            iterations += 1
        for joint in self.joints:
            joint.setVelocity(0)
        timef = self.supervisor.getTime()
        error = np.abs(np.array(target) -
                       self.get_joint_angles()) * 180 / np.pi
        print("Total iterations:", iterations)
        elapsed = timef - time0
        return (
            timef - time0,
            np.max(error),
            np.mean(error),
            (pos, vel, acc, jerk, time_arr),
            end_effector_vel,
```

```
        )

    def move_to_pose(
        self, pos: "np.ndarray", rot: "np.ndarray", euler="XYZ", wrist="down", shoulder="left", duration=None, verbose=False
    ):
        """
        Move to specified position and orientation

        Parameters:
            pos: [x, y, z] coordinates
            rot: [rot_x, rot_y, rot_z] Euler angles
            euler: Euler angle order (default: 'XYZ')
            wrist: 'up' or 'down'
            shoulder: 'left' or 'right'
            duration: time to reach position
            verbose: print error
        """
        T = build_matrix(pos, rot, euler=euler)
        joint_angles = inverse_kinematics(
            T, wrist=wrist, shoulder=shoulder, elbow="up")
        if duration is not None:
            self.move_to_config(target=joint_angles, duration=duration)
        else:
            self.move_to_config(joint_angles)
        if verbose:
            gt = self.get_ground_truth()
            angle_err, pos_err = matrix_error(T, gt)
            print("Angular error:", angle_err, "degrees")
            print("Positional error: ", pos_err, " mm")

    def actuate_gripper(self, close=0, duration=2):
        """
        Actuate gripper to open or close

        Parameters:
            close (int): 0 to open, 1 to close
            duration (int | float): time to close or open
        """
        t0 = self.supervisor.getTime()
        v0 = np.zeros(9)
        vf = np.zeros(9)
        q0 = self.get_finger_angles()
        qf = np.hstack((np.array([lim[close]
                       for lim in self.finger_joint_limits])))
        a0 = np.zeros(9)
        af = np.zeros(9)
        tf = t0 + duration
        A = np.array(
            [
                [1, t0, t0**2, t0**3, t0**4, t0**5],
                [0, 1, 2 * t0, 3 * t0**2, 4 * t0**3, 5 * t0**4],
                [0, 0, 2, 6 * t0, 12 * t0**2, 20 * t0**3],
                [1, tf, tf**2, tf**3, tf**4, tf**5],
                [0, 1, 2 * tf, 3 * tf**2, 4 * tf**3, 5 * tf**4],
                [0, 0, 2, 6 * tf, 12 * tf**2, 20 * tf**3],
            ]
        )
        b = np.array([q0, v0, a0, qf, vf, af])
        x = [np.linalg.solve(A, b[:, i]) for i in range(9)]
        time0 = self.supervisor.getTime()
        iterations = 0
        self.setup_control_mode()
        while self.supervisor.getTime() <= tf:
            t = self.supervisor.getTime()
            for idx, joint in enumerate(self.finger_joints):
                joint.setPosition(
                    x[idx][0]
                    + x[idx][1] * t
                    + x[idx][2] * t**2
                    + x[idx][3] * t**3
                    + x[idx][4] * t**4
                    + x[idx][5] * t**5
                )
            self.supervisor.step(self.timestep)
            iterations += 1
        for i, joint in enumerate(self.joints):
            joint.setPosition(self.finger_joint_limits[i][close])
        timef = self.supervisor.getTime()
        print("Total iterations: ", iterations)
        print(timef - time0)

    def predict_bottle_position(self, show_img=True):
        """
```

```
    Predict bottle position using VGG16 model

    Parameters:
        show_img (bool): show image with predicted position

    Returns:
        (x, y) coordinates of bottle in frame 0
    """
    self.setup_camera()
    img = self.get_image()

    resized_img = resize(img, (224, 224), anti_aliasing=True)

    img_tensor = np.expand_dims(resized_img, axis=0)

    img_tensor = preprocess_input(img_tensor)

    prediction = self.model.predict(img_tensor)
    ximg, yimg = prediction[0][0], prediction[0][1]

    xlim = [40, 465]
    ylim = [81, 395]
    x_real_lim = [-1.1599511371082611, -1.759950096116547]
    y_real_lim = [0.2911156981348095, -0.511156981348095]
    yreal = np.interp(ximg, xlim, y_real_lim)
    xreal = np.interp(yimg, ylim, x_real_lim)
    R0_world = np.array(
        self.supervisor.getSelf().getOrientation()).reshape(3, 3)
    T0_world = np.array(
        self.supervisor.getSelf().getPosition()).reshape(3, 1)
    th0_world = np.hstack(
        (np.vstack((R0_world, np.zeros((1, 3)))), np.vstack((T0_world, 1)))
    )
    Rbottle_world = np.eye(3)
    Tbottle_world = np.array([xreal, yreal, 0.0]).reshape(3, 1)
    th_bottle_world = np.hstack(
        (np.vstack((Rbottle_world, np.zeros((1, 3)))),
        np.vstack((Tbottle_world, 1)))
    )
    th_world_0 = np.linalg.inv(th0_world)
    th_bottle_0 = np.dot(th_world_0, th_bottle_world)
    if show_img:
        ground_truth = self.get_bottle_frame()[:2, 3]
        print("Real position: ", ground_truth)
        print("Predicted position: ", th_bottle_0[:2, 3])
        plt.imshow(img)
        plt.scatter(ximg, yimg, c="r", s=50)
        plt.show()
    return th_bottle_0[0, 3], th_bottle_0[1, 3]

  def get_bottle_frame(self):
      """
      Get bottle frame relative to robot base

      Returns:
          bottle_frame: bottle frame relative to robot base
      """
      Rbottle_world = np.array(
          self.bottle.getOrientation()
      ).reshape(3, 3)
      Tbottle_world = np.array(self.bottle.getPosition()).reshape(
          3, 1
      )
      th_bottle_world = np.hstack(
          (np.vstack((Rbottle_world, np.zeros((1, 3)))),
           np.vstack((Tbottle_world, 1)))
      )
      R0_world = np.array(
          self.supervisor.getSelf().getOrientation()).reshape(3, 3)
      T0_world = np.array(
          self.supervisor.getSelf().getPosition()).reshape(3, 1)
      th0_world = np.hstack(
          (np.vstack((R0_world, np.zeros((1, 3)))), np.vstack((T0_world, 1)))
      )
      th_world_0 = np.linalg.inv(th0_world)
      th_bottle_0 = np.dot(th_world_0, th_bottle_world)
      return th_bottle_0
```