Mechatronic Systems Laboratory

Manipulator Robot - Pick and Place Tasks

UNIVERSITÄT SIEGEN

Lab report submitted by **Group 9**
Winter Semester 2023-24

Divya Prakash Biswas 1701940
Hafsa Siddiqui 1702062
Pragya Mahajan 1700774
Ramesh Rahul Davanam 1701333
Shantanu Sham Shirsath 1699551

Contents

List of Figures

1 Introduction

This article reports the solution of the given **"Mechatronic Systems"**project task, that has to be carried out on a **LEGO Mindstorms EV3** manipulator. We are enrolled for the project as **Group 9**, share our solution methodology, understanding and findings through this report.

As it was instructed to use MATLAB as a programming software for the robot, we have developed a MATLAB package containing multiple functions and scripts to generate desired set of robot behaviors. In the following sections we will first discuss about the derived kinematic model of the robot and then the implementation of that model to perform specified tasks in the MATLAB.

2 Kinematics

A Robot Manipulator is a robot Arm made of multiple links and different types of joints. Links Are rigid components that connect different parts of the robot. Joints are used to connect two links and provide a relative motion between the links. The LEGO Mindstorm EV3 robot manipulator for pick and place operation is shown in Figure 1 and its Kinematic representation in Figure 2.

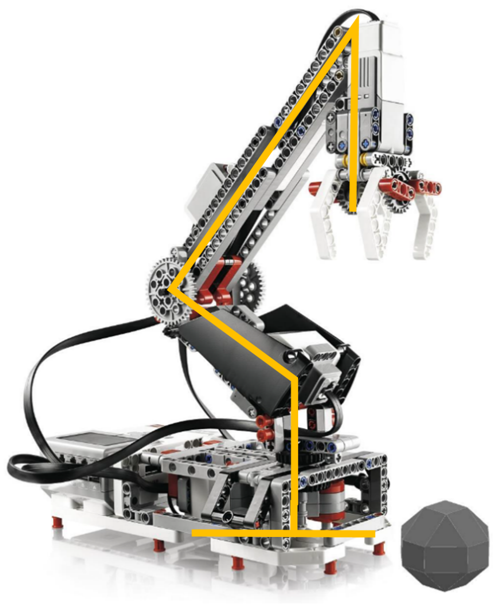
Figure 1: EV3 robot

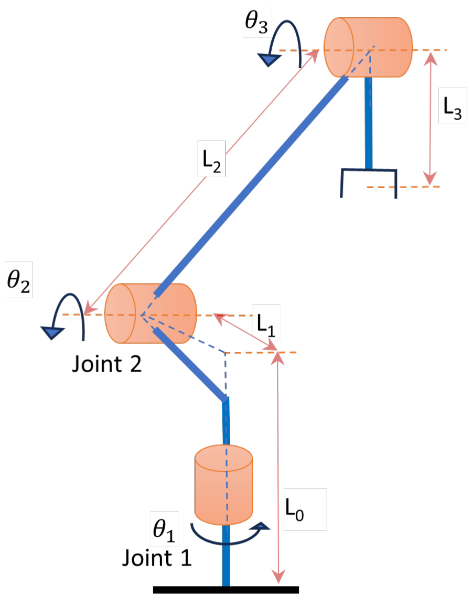

Figure 2: Kinematic Model

2.1 Forward Kinematics

We have developed a kinematic model of our robot and assigned frames to joints according to Denavit-Hartenberg(DH) rules, listed here.

- Assign Z axes as per the direction of rotation in revolute joints and in the direction of motion in prismatic joints.
- Assign X_0 and Y_0 arbitrarily so as to satisfy right-hand rule.
- Assign X_i such that is common normal to Z_i and Z_{i-1} and intersecting both. Assign Y_i as per right-hand rule.

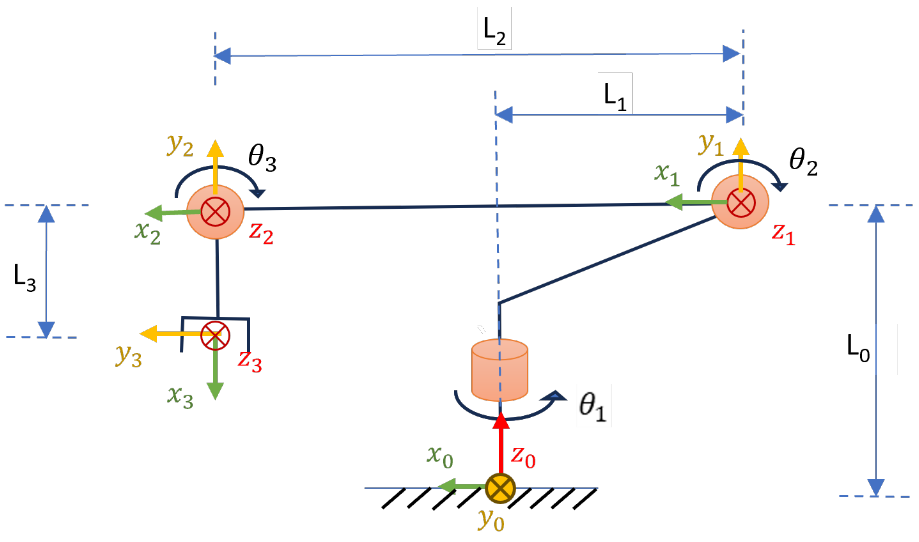

Figure 3: Frame Assignments

Denevit-Hartenberg(DH) Parameters and are used to calculate the Transformation matrix between two points, i and $i-1$ of the link. Description of the DH parameters is given below

- Angle rotated along Z_{i-1} axis so as to align X_i and X_{i-1} (rotation of $i-1$ frame).
- Perpendicular distance between X_i and X_{i-1} along Z_{i-1} axis.
- Common Normal distance between Z_i and Z_{i-1} measured along X_i.
- Angle rotated along X_i so as to align Z_i with Z_{i-1} (rotation of $i-1$ frame).

Once the DH parameters are found, we can compute the frame transformations from the fixed frame to any moving frame. The DH parameters for our EV3 robot can be found in Table 1.

n	θ	d	a	α
1	θ_1	L_0	-L_1	$\frac{\pi}{2}$
2	θ_2	0	L_2	0
3	$\theta_3 - \frac{\pi}{2}$	0	L_3	0

Table 1: DH Parameter Table

In *Forward Kinematics* our goal is to compute the end-effector(gripper) position with respect to the fixed frame. We have created a MATLAB script `Chuka_robot.m`, that helps us to compute *Forward Kinematics* of the robot.

Note: We have assigned three revolute joints in the robot model. But we know that there are only two actuators on the robot at Joint1 and Joint2. The third revolute joint is actually kinematically constrained and depends on the Joint2, due to the four-bar mechanism. So we mathematically constrain the third revolute joint by θ_3 = -θ_2.

From the task description, we know that the robot should go to *HomeConfiguration* after performing each task. We also define a configuration when both joint angles are zero. We are interested to know the gripper positions at these two special configurations because these values are used in our implementation.

- *ZeroConfiguration*: $\theta_1 = 0$ and $\theta_2 = 0$.
- *HomeConfiguration*: $\theta_1 = 0$ and $\theta_2 = 40$ degrees (maximum limit and pressing the touch sensor).

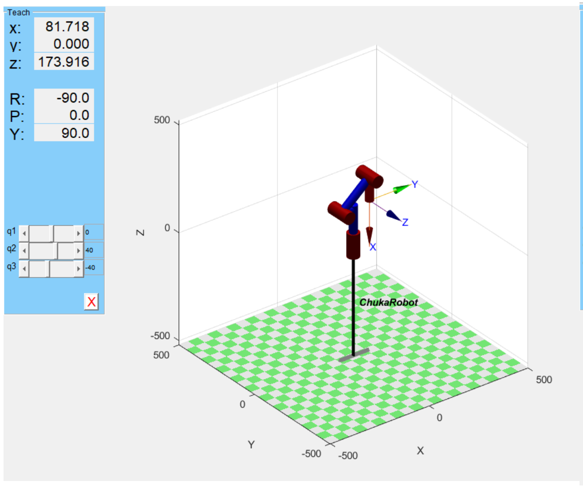
Teach
x: 81.718
y: 0.000
z: 173.916
R: -90.0
P: 0.0
Y: 90.0
q1 0
q2 40
q3 -40

Figure 4: Home Configuration

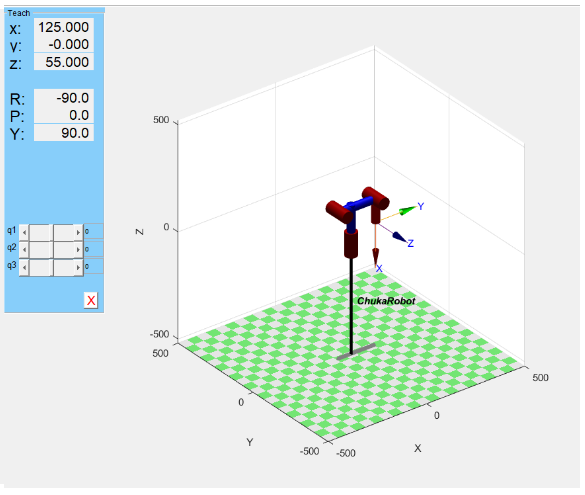
Teach
x: 125.000
y: -0.000
z: 55.000
R: -90.0
P: 0.0
Y: 90.0
q1 0
q2 0
q3 0

Figure 5: Zero Configuration

We noted the X and Y coordinates of the gripper from the *ZeroConfiguration* and Z coordinates of the gripper from the *HomeConfiguration*(`Hmax` = 174).

2.2 Inverse Kinematics

In *Inverse Kinematics* we calculate the required joint angles for a specific gripper position in 3D space. We assign coordinates to the desired gripper position as (X_p, Y_p, Z_p) with respect to frame 0. We have used *Geometric Approach* to calculate joint angles. From Figure 6 we can find equations for calculating θ_1 and θ_2.

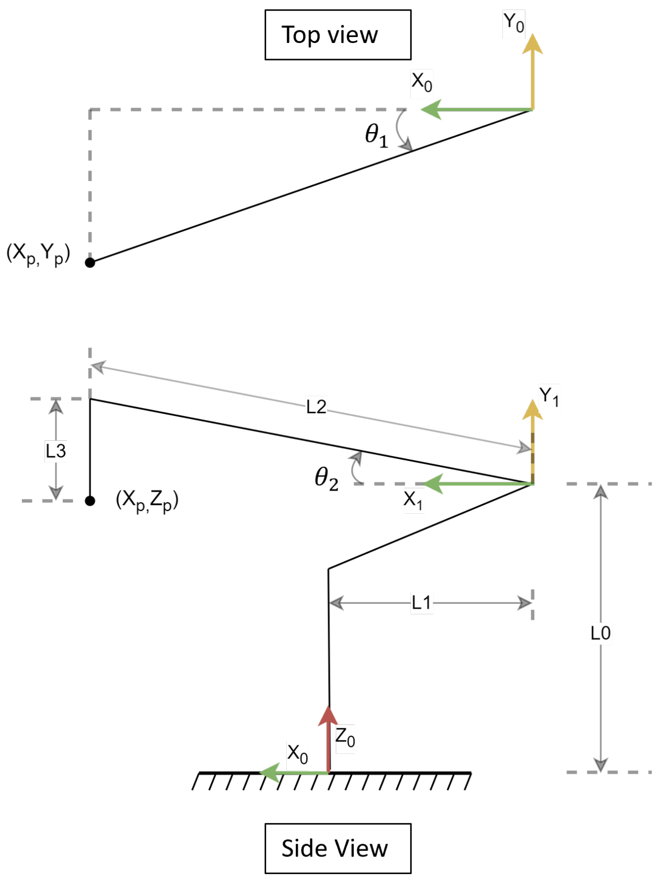

Figure 6: Inverse Kinematics

$$\tan(\theta_1) = \frac{Y_p}{X_p}$$

$$\theta_1 = atan2(Y_p, X_p) \tag{1}$$

$$Z_p = L_0 + L_2 \sin(\theta_2) - L_3$$

$$\sin(\theta_2) = \frac{Z_p + L_3 - L_0}{L_2} = \eta$$

$$\cos(\theta_2) = \sqrt{1 - \eta^2}$$

$$\theta_2 = atan2(\eta, \sqrt{1 - \eta^2}) \tag{2}$$

Note: The equation 1 and 2 are totally independent and can be computed separately. Hence for a given desired gripper position (X_p, Y_p, Z_p), we follow the sequence:

- Calculate θ_1
- Move Joint1
- Calculate θ_2
- Move Joint2

3 Implementation

In our MATLAB package `main_chuka.m` is the top-level script that can be run directly. This script calls other scripts in sequence to **Initialize the Workspace**, **Initialize the Robot**, **Calibration** etc. In this section we have discussed about these behaviors and mentioned the responsible script or functions for them. The Figure 7 shows the step-wise break down of the `main_chuka.m` script. First three tasks are crucial and must be performed before using any function from the package. The tasks shown in blue are the manipulation tasks and they are handled by individual scripts.

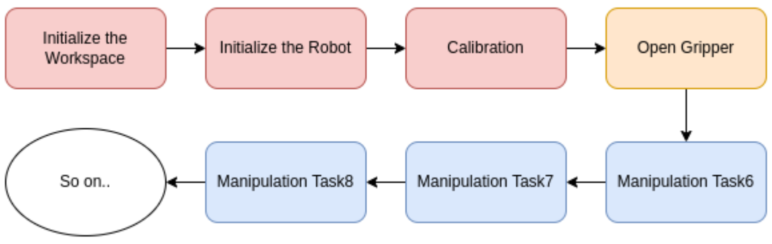

Figure 7: Execution flow of main script

3.1 Initialize the Workspace

The script `init_workspace.m` serves the purpose of defining the 2D locations of three stations A, B and C within the robot workspace. These stations represent specific areas where activities or tasks will be performed by the robot. These station locations are utilized as reference points for navigation, task allocation within the workspace.

3.2 Initialize the Robot

The script `init_chuka.m` declares all the objects and variables necessary for the robot. It sets up the hardware components of the LEGO Mindstorms EV3 robot, including sensors, motors, and their respective port assignments. It also defines parameters such as gear ratios and motor speeds. These initialization are essential for controlling the robot and executing various tasks such as navigation, sensing and manipulation.

3.2.1 Controller Object

The LEGO Mindstorms EV3 controller is the CPU of the robot that is connected with other peripherals such as motors and sensors. We connect the robot CPU to MATLAB via creating a `legoev3` object passing the communication media 'usb' as a parameter. We have named that object as `chuka_robot` and used it throughout our program as the main robot instance.

3.2.2 Sensor and Motor Ports

Sensor and motor ports are assigned specific numbers and letters for identification. This ensures that MATLAB can communicate with the correct hardware components connected to the robot. The port numbers to the sensors and motors are connected to the robot CPU. `tSen1Port` and `tSen2Port` represent ports for two touch sensors, while `heightSensorPort` represents the port for the ultrasonic sensor. `motor1Port` and `motor2Port` represent ports for two motors controlling different joints, and `gripperPort` represents the port for the motor controlling the gripper mechanism.

3.2.3 Sensor Objects

Touch sensors are used to detect physical contact. In the robot they are placed to detect the maximum reach of two joints of the robot. An Ultrasonic sensor is used to measure distance using ultrasonic waves. Our robot uses the ultrasonic sensor to measure the height of the robot end effector. `touchsensor1` and `touchsensor2` represent objects for the two touch sensors, and `height_sensor` represents an object for the ultrasonic sensor.

3.2.4 Motor Objects

Motors are used to actuate different joints of the robot. In our robot, `motor1` and `motor2` represents motor objects that control the movement of joints, while `gripper` represents the motor object that actuates the opening and closing of the gripper mechanism of our LEGOEV3 robot.

3.2.5 Other Parameters

Gear ratios are necessary to get the speed and torque of the joints attached to motors. The Joint1 of the robot uses a gear reduction of 3:1 and Joint2 uses a gear reduction of 5:1. `GRjoint1` defines gear reduction for joint 1 and `GRjoint2` defines gear reduction for Joint2. We have also defined the default speed at which the gripper motor operates.

3.3 Calibration

A robot's sensors, actuators, and other parts must have their settings adjusted and fine-tuned to meet the required standards. This process is known as calibration. Robot deployment and maintenance require calibration as a basic process, which improves the robots' accuracy, dependability, safety, and general performance across a range of applications. Keeping this ideology in mind we have also included a mandatory calibration process before performing any task. In the script `calibration_sequence.m` we run both the motors in a direction that would lead the joints to press the touch sensors. The moment the joints hit the sensors, we reset the motor encoder values to zero. This process makes our implementation more robust and hardware-independent.

The calibration is performed in the sequence of `motor2`, `motor1` and then gripper. The `motor2` is calibrated before motor1 because if `motor1` is calibrated first without calibrating `motor2` then we

may run a risk of hitting any of the stations along the workspace of the robot. The `motor2` starts to move with defined speed `motor_speed_calib` until `touchsensor2` is pressed, indicating that the actuator has reached it's limit. Then we reset the encoder value of the motor. Similarly, `motor1` is started and set to move in the forward direction with defined speed `motor_speed_calib` until `touchsensor1` is pressed and then we reset the encoder.
The gripper motor can't be calibrated as other motors as it doesn't have any touch sensor attached. So we rotate the gripper motor in forward direction at a constant speed for 3 seconds to makes sure that our gripper is completely closed before resetting it's encoder value.
Additionally, between each calibration step,function `pause()` have been inserted to make sure that the motor has finished rotating and the joints have reached a fully stable position and not oscillating. Short sounds [`beep()`] are produced to indicate that a step has concluded.

3.4 Gripper actions

Gripper actions are implemented by two MATLAB functions: `gripper_open()` and `gripper_close()`. Both the functions takes gripper motor object and speed as arguments. Because of the calibration process now we just need to rotate the gripper motor in specific direction to specific angular position in order to open or close it. So, to open the gripper we rotate the motor in reverse direction with negative speed until the encoder reads -90. Similarly the gripper motor is rotated in forward direction with positive speed until the encoder reads -20, to close the gripper. After both the operations we set the motor speed to zero. Once tasks are completed, our robotic system reliably returns to its initial position – a homing procedure defined by a structured homing script.

3.5 Joint Control

Achieving precise motor positions is managed through the integration of a PID controller within the `setMotorPos()` function. This function is designed to control the position of two motors using a proportional-integral-derivative (PID) controller. We decided to command the joints to rotate based on our derived kinematic model. The joint angles and motor angles differ by a factor of gear ratio. The `setMotorPos()` function takes three arguments: `motor_obj` (the motor object), `motor_no` (indicating which motor to control, either 1 or 2), and `des_angle` (desired joint target angle). We mapped the desired joint angles(software) to motor angles(hardware) by mapping functions `motor1_map` or `motor2_map` and multiplying them with respective gear ratios.
The `setMotorPos()` function sets PID controller gains and gear ratios (`GR_joint`) based on the specified motor number. The PID controller is implemented within a while loop, where the motor's current position is continuously read and compared to the desired position. The loop iterates until the absolute position error is within a specified tolerance band (in our case, 2). The PID control effort is computed based on the error, error integral and derivative.

```
u_out = KP * er + KI * er_sum + KD * (er - er_prev)
```

The above control algorithm calculates the required motor speed (`u_out`) as a weighted sum of three components: the proportional term (`KP * er`), responding to the current error; the integral term (`KI * er_sum`), addressing cumulative error over time; and the derivative term (`KD * (er - er_prev)`), considering the rate of change of the error. After tuning, the PID gains were determined as follows. For `motor1`: `KP = 0.3, KI = 0.002, KD = 5` and for `motor2`: `KP = 0.2, KI = 0.01, KD = 30`. After the motor reaches to the desired position, the motor speed is set to `hold_power`. This value is zero for joint 1 and non-zero for joint 2 to maintain the arm position against gravity.

3.5.1 Position mapping

As we compute the desired joint angles from the inverse-kinematic model, we refer joint angles as software angles and the actual motor angles as hardware angles. `motor_map` is a mapping function that translates software angles to corresponding hardware values for a specific motor. It takes a software angle (`software_ang`) as an input and utilizes linear interpolation to map it to a hardware angle. The function is defined within a specified range for both hardware and software angles.
For `motor1`: the software angle between -100 to 100 degrees range is mapped to hardware angle between a range of 0 to 200 degrees.
For `motor2`:the software angle between 40 to -30 degrees range is mapped to hardware angle between a range of 0 to 70 degrees.

3.5.2 Limit Power

The `limitPow()` function restricts the input motor power within a predetermined range. Given a motor power input (in), the function compares its absolute value to a specified limit (`limit_value`). If within the limit, the input is returned unchanged. If exceeding the positive limit, the output is constrained to the positive limit, and if below the negative limit, the output is regulated to the negative limit. The `limit_value` for both of the motors was chosen to be 70, because this speed seemed more than sufficient to perform the tasks.

3.6 Pick and Place Task

The robot was explicitly designed to execute the task of picking up a ball from one station and placing it at another. Once the foundational functions for basic operations were established, our focus shifted to integrating these functions to fulfill the overall objective. Consequently, we created the `placeToStation()` and `pickFromStation()` functions to facilitate the picking and placing behavior respectively. We have also developed homing behavior in script `homing.m`, to set robot configuration to $HomeConfiguration$(refer Figure 4).

As we have mentioned in section 2.2, calculation of desired angle for Joint1 and Joint2 are completely independent mathematically. Calculation of the desired Joint1 angle(θ_1) only needs X and Y coordinates and desired Joint2 angle(θ_2) requires Z coordinate of the desired 3D position.
For both pick and place behavior, we want our robot to **dynamically adjust** according the station provided. So we need to estimate the height of station(the Z coordinate of the target 3D position) in both pick and place task. The height estimation is one of the crucial tasks because by this we calculate the desired Joint2 angle(θ_2). The height is estimated using the ultrasonic sensor readings and previously calculated parameter `Hmax` and measured parameter `offset`. Modified $InverseKinematics$ algorithm:

- Calculate θ_1
- Move Joint1
- Estimate station height
- Calculate θ_2
- Move Joint2

3.6.1 Station Height Estimation

We always read the ultrasonic sensor when the Joint2 is at maximum position(pressing the limit switch). From our forward kinematics model we know that at this configuration, the gripper center stays `Hmax = 174mm` high from the fixed base frame. We also know the vertical distance between gripper center and ultrasonic sensor, `offset = 51mm`. We store the measurement of the ultrasonic sensor in a variable `h`. Then we estimate the height of the station by using following equations. Refer Figure 8 for better clarification.

$$H_o = Hmax + offset$$

$$Z_p = H_o - h$$

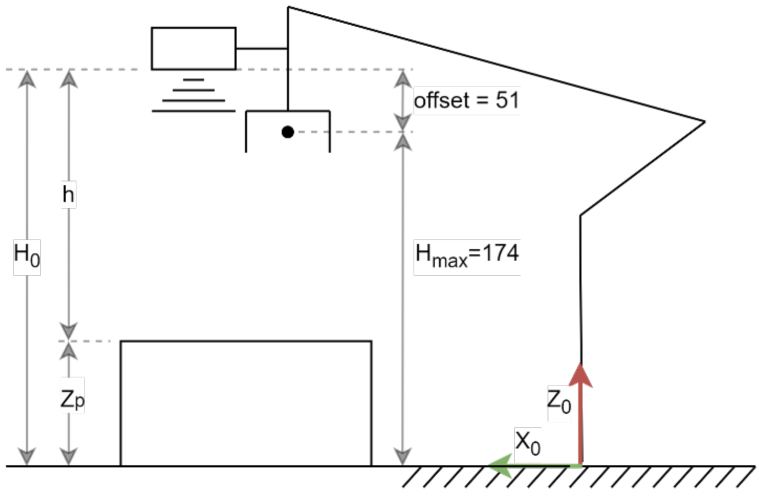

Figure 8: Height Estimation

3.6.2 Picking Behavior

The `pickFromStation()` function facilitates the ball-picking process in the robotic system with two motors, a gripper, and the ultrasonic sensor. It begins by calculating the desired angle for the Joint1 based on the target station coordinates and sets the `motor1` to this position by using function `setMotorPos()`. A brief pause of 3 seconds is given to settle the oscillation caused by to motion of Joint1. Then the measurement is taken from the ultrasonic sensor, and the height of the station is computed. Utilizing this information, the function calculates the desired angle for the Joint2, adjusts its position, and closes the gripper to grab the ball. Then Joint2 rotates to it's initial position. Refer to Figure 9.

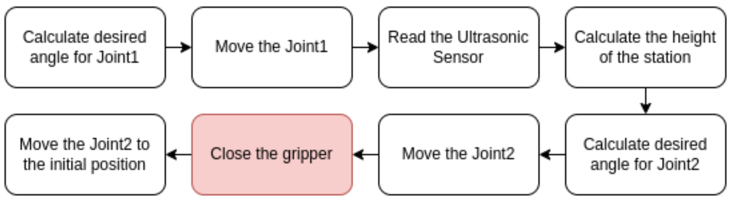
Calculate desired angle for Joint1
Move the Joint1
Read the Ultrasonic Sensor
Calculate the height of the station
Calculate desired angle for Joint2
Move the Joint2
Close the gripper
Move the Joint2 to the initial position

Figure 9: Picking Behavior

3.6.3 Placing Behavior

The `placeToStation()` function is designed to place a ball at a specified station using the robot. It needs arguments as two motors (`motor1` and `motor2`), a gripper with a defined speed (`gripper` and `gripper_speed`), a ultrasonic sensor and the coordinates of the target station (`station`). This function complements the `pickFromStation()` function, sharing a similar structure for efficient and controlled ball placement. As it can be seen in Fig9 and Fig10, both behaviors are similar in sequence, but differ only the gripper action.

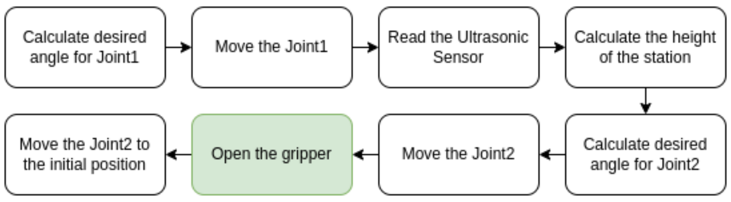
Calculate desired angle for Joint1
Move the Joint1
Read the Ultrasonic Sensor
Calculate the height of the station
Calculate desired angle for Joint2
Move the Joint2
Open the gripper
Move the Joint2 to the initial position

Figure 10: Placing Behavior

3.7 Manipulation Task

The manipulation of the given object(ball) is achieved by combination of previously described cellular behaviors. In the project objective, task6 to 11 are manipulation tasks. For each of them we have created separate script. The script `task8.m` is for task8, for example. All the manipulation tasks are similar in structure. So describing one is enough to understand remaining manipulation tasks. As per instructions, we perform homing by `homing.m` first. Then pick from a given station by using function `pickFromStation()`. Then we place the ball to desired station by function `placeToStation()`. After, we again move the robot to $HomeConfiguration$ by `homing.m` script. Here in Figure 11 execution of task6 is explained by a flow chart.

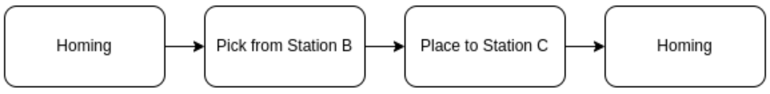

Figure 11: Manipulation Task6