Experiment No: 1
Calibration of Sensor

Objective

1. To calibrate HC-SR04 ultrasonic sensor by interfacing it to an Arduino.
2. To display the calibrated data as a ROS message.

Theory

The ultrasonic sensor working principle is the same as the object detection system of a bat. Also, we can say that it works on the same principle as a radar system. The ultrasonic or HC-SR04 module has two main parts: the Transmitter (TX) and the Receiver (RX). The transmitter sends Ultrasonic (US) sound and the Receiver receives the US sound.

First of all, we need to set the Trig pin on a High State for 10 µs (microseconds) to generate the ultrasonic sound. Then it will send out an 8-cycle sonic burst which will travel at the same speed of sound. If the 8-cycle waves fall on any object and it bounces back from the object surface, then it's collected by the receiver part of the module. As a result, the ultrasonic sensor echo pin produces a high pulse output. The output pulse duration is the same as the time difference between transmitted ultrasonic bursts and the received echo signal.

$$s = \frac{v \times t}{2}$$

Where s is the distance between the sensor and object, v is the speed of sound in air ($v = 0.034$cm/µs or 340 m/s), and t is the time sound waves take to bounce back from the object's surface. We need to divide the distance value by 2 because time will double as the waves travel and bounce back from the initial point.

One Point Calibration

One point calibration is the simplest type of calibration. If your sensor output is already scaled to useful measurement units, a one point calibration can be used to correct for sensor offset errors in the following cases:

- Only one measurement point is needed. If you have an application that only requires accurate measurement of a single level, there is no need to worry about the rest of the measurement range.
- The sensor is known to be linear and have the correct slope over the desired measurement range. In this case, it is only necessary to calibrate one point in the measurement range and adjust the offset if necessary.

A one point calibration can also be used as a "drift check" to detect changes in response and/or deterioration in sensor performance.

Interfacing Arduino with HC-SR04

Procedure

1. Interface the ultrasonic sensor to Arduino.

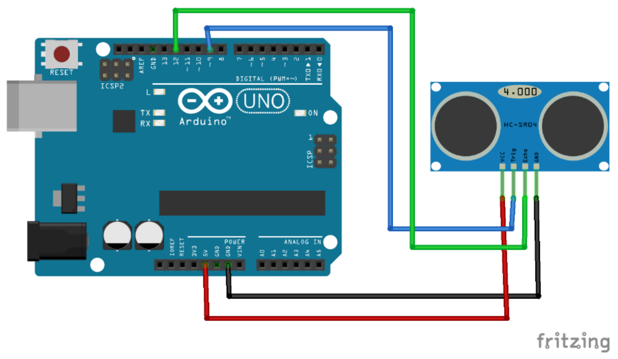


Figure 1: Interfacing Arduino with HC-SR04

2. Keep an obstacle at a marked distance.
3. Log the distance value recorded via the serial monitor.
4. Repeat the process for obstacles placed at different known distances.
5. Plot a graph with actual distance measured using a scale vs. the distance displayed by the serial data from the sensor.

To display the sensor data in ROS terminal

1. Open a terminal in Ubuntu and run `roscore`.
2. Open a new tab, and run the code `rosrun rosserial_python serial_node.py /dev/ttyUSB0`
3. From a new terminal, use `rostopic echo chatter` to display the sensor output data.

Calculation

Code

Sample Arduino code for interfacing HC-SR04 ultrasonic sensor:

```
#include <ros.h>
#include <std_msgs/Int32.h>

ros::NodeHandle nh;
std_msgs::Int32 str_msg;
ros::Publisher chatter("chatter", &str_msg);

#define echoPin 12 // attach pin D2 Arduino to pin Echo of HC-SR04
#define trigPin 9 //attach pin D3 Arduino to pin Trig of HC-SR04

// defines variables
long duration; // variable for the duration of sound wave travel
int distance; // variable for the distance measurement

void setup() {
        pinMode(trigPin, OUTPUT); // Sets the trigPin as an OUTPUT
        pinMode(echoPin, INPUT); // Sets the echoPin as an INPUT
        Serial.begin(9600); // Serial Communication
        nh.initNode();
        nh.advertise(chatter);
}
```

```
void loop() {
        // Clears the trigPin condition
        digitalWrite(trigPin, LOW);
        delayMicroseconds(2);
        // Sets the trigPin HIGH (ACTIVE) for 10 microseconds
        digitalWrite(trigPin, HIGH);
        delayMicroseconds(10);
        digitalWrite(trigPin, LOW);

        duration = pulseIn(echoPin, HIGH);
        // Calculating the distance
        distance = duration * 0.034 / 2; //
        Serial.print("Distance: ");
        Serial.print(distance);
        Serial.println(" cm");
        str_msg.data = distance;
        chatter.publish(&str_msg);
        nh.spinOnce();
        delay(1000);
}
```

Observation

Sl no:	Actual distance	Measured distance
1		
2		
3		
4		
5		
6		
7		

Result

Experiment No: 2
Trajectory Planning

Objective

To generate a linear trajectory with a parabolic blend for a joint of robot manipulator within a given time frame using angular velocity and angular acceleration as inputs.

Theory

- **Linear Trajectory**: The simplest form of trajectory planning where the robot moves in a straight line from the initial position θ_i to the final position θ_f. While straightforward, it can lead to abrupt starts and stops.
- **Parabolic Blend**: To smooth out the abrupt transitions at the beginning and end of the linear trajectory, a parabolic blend is introduced. This blend creates a smooth, curved transition that prevents jerks and ensures a more natural motion.

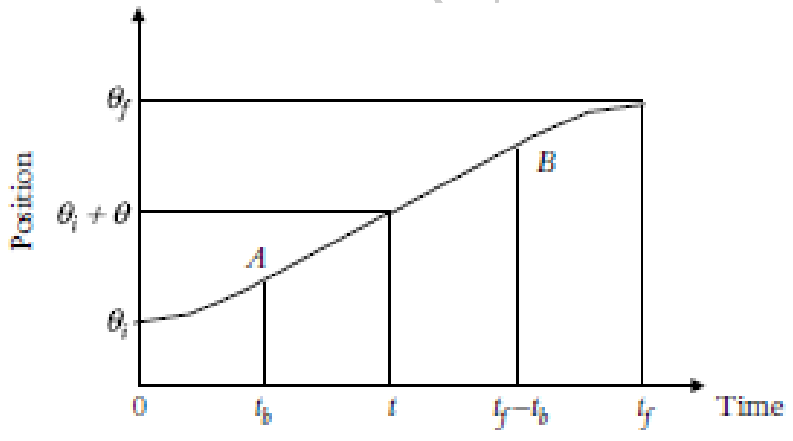


Figure 1: Linear Trajectory with Parabolic Blend

- **Blending Time** (t_b): The time during which the parabolic blend occurs. It ensures a smooth transition from rest to motion and vice versa. The blending time can be calculated using the following formula:

$$t_b = \frac{\theta_i - \theta_f + \omega t_f}{\omega} \tag{1}$$

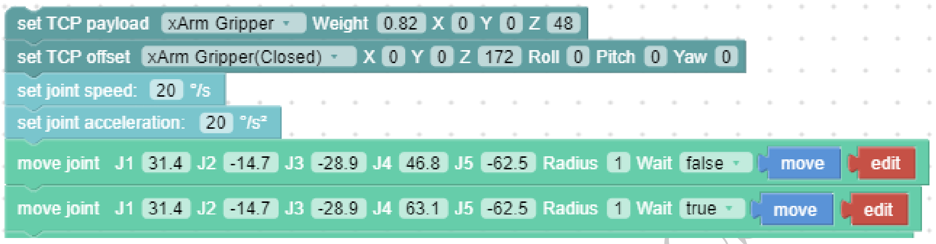
  Where:

  - t_b = Blending time
  - θ_i = Initial joint position
  - θ_f = Final joint position
  - ω = Angular velocity
  - t_f = Total time for the movement

- **Angular Velocity** (ω): The rate at which the joint moves. It plays a crucial role in determining the blending time and ensuring the motion is executed within the desired timeframe.

Procedure

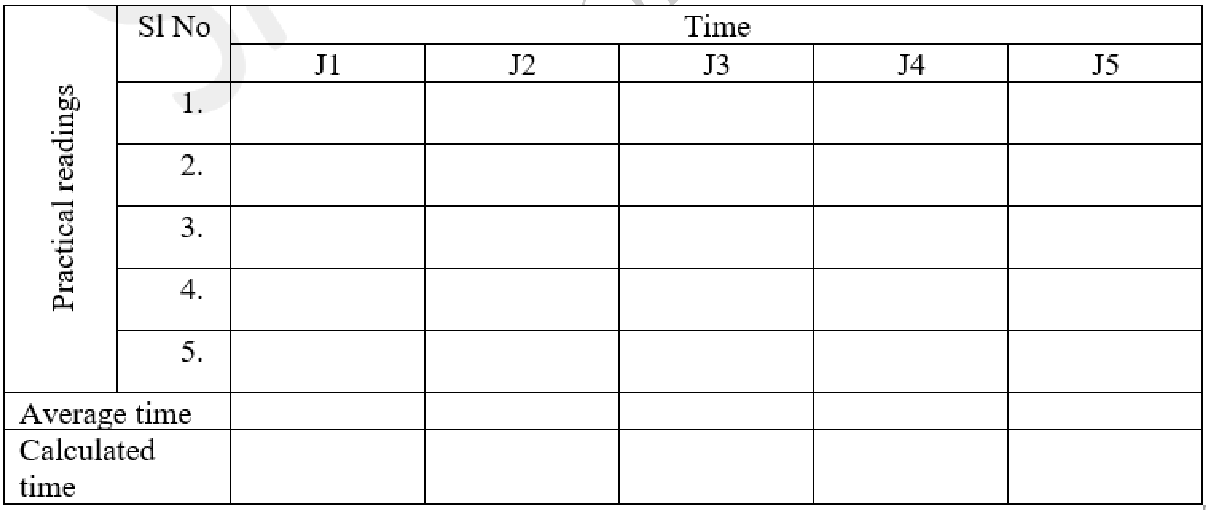
1. **Determine Parameters**: Identify the initial and final positions (θ_i and θ_f), the desired total time (t_f), and the angular velocity (ω).
2. **Calculate Blending Time**: Use the blending time formula to determine t_b. This helps in defining the duration for the parabolic transition at the start and end of the motion.
3. **Design Trajectory**: Create the trajectory by combining the linear segment with parabolic blends at both ends. The robot starts with a parabolic acceleration phase, transitions into a linear motion, and finally decelerates parabolically to a stop.
4. **Execute and Monitor**: Implement the planned trajectory on the robot Continuously monitor and adjust the parameters if necessary to ensure smooth and precise movement.



set TCP payload xArm Gripper Weight 0.82 X 0 Y 0 Z 48
set TCP offset xArm Gripper(Closed) X 0 Y 0 Z 172 Roll 0 Pitch 0 Yaw 0
set joint speed: 20 °/s
set joint acceleration: 20 °/s²
move joint J1 31.4 J2 -14.7 J3 -28.9 J4 46.8 J5 -62.5 Radius 1 Wait false move edit
move joint J1 31.4 J2 -14.7 J3 -28.9 J4 63.1 J5 -62.5 Radius 1 Wait true move edit

Figure 2: Sample block code in XARM5 studio

Observations



	Sl No	Time				
		J1	J2	J3	J4	J5
Practical readings	1.					
	2.					
	3.					
	4.					
	5.					
Average time						
Calculated time						

Figure 3: Trajectory Planning

Results:

Experiment No: 3
Forward Kinematics

Objective:

Obtain the DH parameters of X-arm 5 lite and forward kinematics.

Theory:

The XARM5 with DH parameters is given as follows. Based on the DH convention, the transformation

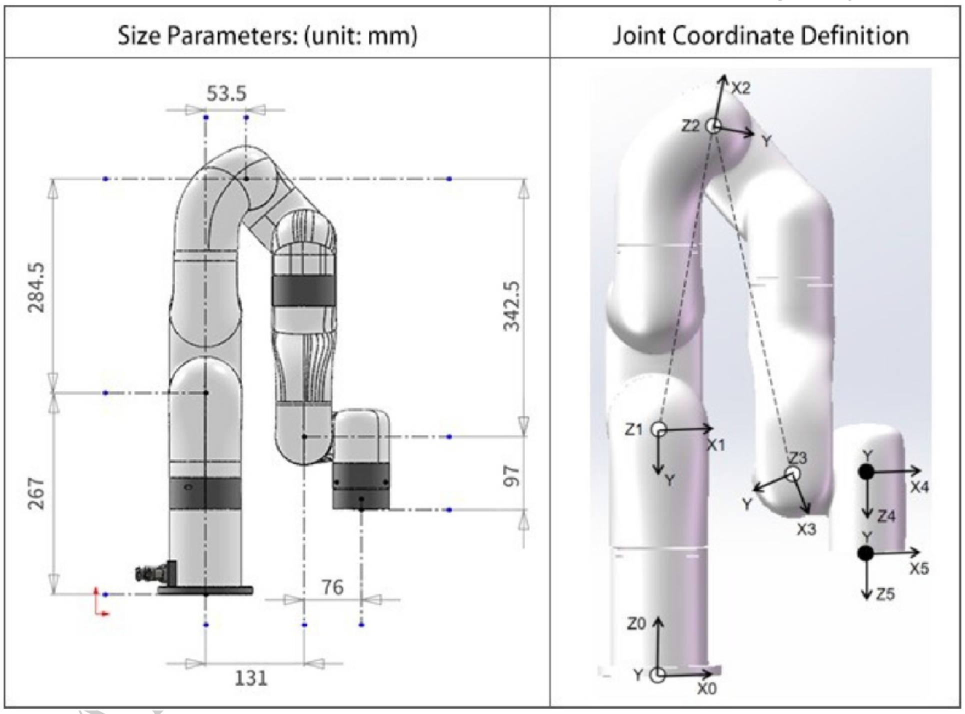
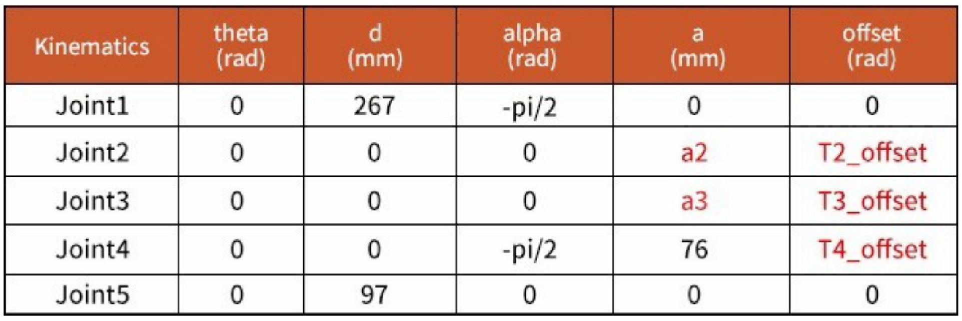


Figure 1: Figure 1: X-arm 5 lite Robot



Kinematics	theta (rad)	d (mm)	alpha (rad)	a (mm)	offset (rad)
Joint1	0	267	-pi/2	0	0
Joint2	0	0	0	a2	T2_offset
Joint3	0	0	0	a3	T3_offset
Joint4	0	0	-pi/2	76	T4_offset
Joint5	0	97	0	0	0

Figure 2: Figure 1:DH parameters

matrix from joint i to joint $i+1$ is given by:

$$ {}^{i+1}_{i}T = A_{n+1} $$

$$ = \mathrm{Rot}(z, \theta_{n+1}) \cdot \mathrm{Trans}(z, d_{n+1}) \cdot \mathrm{Trans}(x, a_{n+1}) \cdot \mathrm{Rot}(x, \alpha_{n+1}) $$

$$ = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1) $$

Where $\sin\theta_i = \sin\theta_i$, $\cos\theta_i = \cos\theta_i$, $\sin\alpha_i = \sin\alpha_i$, $\cos\alpha_i = \cos\alpha_i$, $\sin(\theta_i + \theta_j + \theta_k) = \sin(\theta_i + \theta_j + \theta_k)$ and $\cos(\theta_i + \theta_j + \theta_k) = \cos(\theta_i + \theta_j + \theta_k)$.

For a 5 DoF robot,

$$ {}^{0}T = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_1 A_2 A_3 A_4 A_5 $$

The above equation will give expressions for the end effector position and orientation.

NB: Joint variable θ will be updated according to the table below after considering offset. Use the new value with offset for Arm matrix calculation.

Joint variables (degree)	Joint variable (radian)	Offset (radian)	Joint variable + offset (radian)
$J_1 = \theta_1$	$\theta_1(rad)$		$\theta_1(rad)$
$J_2 = \theta_2$	$\theta_2(rad)$	-1.38	$\theta_2(rad) - 1.38$
$J_3 = \theta_3$	$\theta_3(rad)$	2.73	$\theta_3(rad) + 2.73$
$J_4 = \theta_4$	$\theta_4(rad)$	-1.35	$\theta_4(rad) - 1.35$
$J_5 = \theta_5$	$\theta_5(rad)$		$\theta_5(rad)$

Find $[p_x\ p_y\ p_z]^T$ using the above table and DH table by substituting in (1).

Procedure:

1. Find DH parameters.
2. Open MATLAB and prepare code for forward kinematics calculation.
3. Input all DH values and calculate each transformation matrix.
4. Calculate the Arm equation from above calculated matrices.
5. Display the result.
6. Turn On the X-arm and open the X-arm studio.
7. Adjust the slider to set joint parameters as in the table.
8. Run the robot.
9. Observe the end effector position of the real robot, with input joint parameters as in the calculation table.
10. Compare the calculated value with the observed value with the real robot.

Joint variables (degree)	Joint variable (radian)	Offset (radian)	Joint variable + offset (radian)
J_1	θ_1		θ_1
J_2	θ_2	-1.38	$\theta_2 - 1.38$
J_3	θ_3	2.73	$\theta_3 + 2.73$
J_4	θ_4	-1.35	$\theta_4 - 1.35$
J_5	θ_5		θ_5

Calculation:

$$D_1 =$$

$$D_2 =$$

$$D_3 =$$

$$D_4 =$$

$$D_5 =$$

$$DHM =$$

Find $[p_x\ p_y\ p_z]^T$ using the above table and DH table by substituting in (1).

Observation:

Calculated value =

Observed value =

Python Code:

```
import numpy as np
import math

def create_dh_matrix(alpha, a, theta, d):
return np.matrix([
[np.cos(theta), -np.cos(alpha) * np.sin(theta), np.sin(alpha) * np.sin(theta), a * np.cos(theta)],
[np.sin(theta), np.cos(alpha) * np.cos(theta), -np.sin(alpha) * np.cos(theta), a * np.sin(theta)],
[0, np.sin(alpha), np.cos(alpha), d],
[0, 0, 0, 1]
])

# Set print options for better readability
np.set_printoptions(formatter={'float': '{: 0.1f}'.format})

# Main program
if __name__ == "__main__":
d1 = 267
d2 = 0
d3 = 0
d4 = 0
d5 = 97
# Real theta values are to be added with these default values and should be in radians
```

```
theta1 = 0
theta2 = -0.7749262
theta3 = -0.577704
theta4 = 1.35263
theta5 = 0

a1 = 0
a2 = 289.48866
a3 = 351.158796
a4 = 76
a5 = 0

alpha1 = -math.pi / 2
alpha2 = 0
alpha3 = 0
alpha4 = -math.pi / 2
alpha5 = 0

D1 = create_dh_matrix(alpha1, a1, theta1, d1)
D2 = create_dh_matrix(alpha2, a2, theta2, d2)
D3 = create_dh_matrix(alpha3, a3, theta3, d3)
D4 = create_dh_matrix(alpha4, a4, theta4, d4)
D5 = create_dh_matrix(alpha5, a5, theta5, d5)

DHM = np.dot(np.dot(np.dot(np.dot(D1, D2), D3), D4), D5)
pm = np.array([[0], [0], [0], [1]])
pf = np.dot(DHM, pm)

print("Result: \n", pf)
```

Result:

Experiment No: 4
Inverse Kinematics

Objective:

To determine the inverse kinematics of the X-arm 5 lite robot.

Theory

Inverse kinematics (IK) is a critical concept in robotics and computer animation, aiming to compute the joint parameters that result in a desired position and orientation of a robot's end effector. While forward kinematics derives the end effector's position from known joint parameters, inverse kinematics works in reverse.

Key Concepts

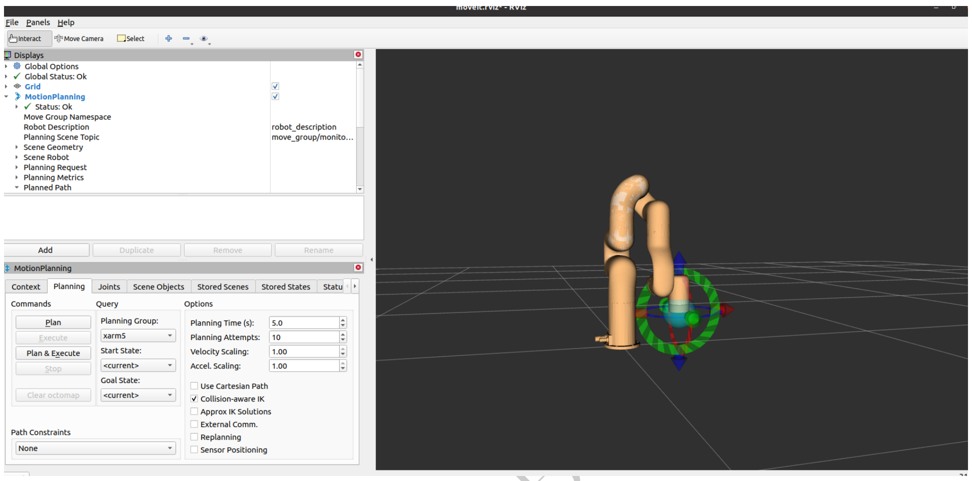
- **End Effector**: The end effector, such as a gripper or tool, interacts with the environment. The goal of IK is to position this end effector precisely.
- **Kinematic Chain**: A series of links connected by joints forms the robot arm. Each joint contributes to the overall position of the end effector.
- **Transformation Matrices**: Using Denavit-Hartenberg (DH) parameters, homogeneous transformation matrices represent the position and orientation of the end effector relative to the robot's base frame.
- **DH Parameters**:
  - a: Link length
  - : Link twist
  - d: Link offset
  - : Joint angle
- **IK Algorithms**: These algorithms compute joint angles for a desired end effector position. Common methods include analytical solutions, numerical methods, and iterative approaches like the Jacobian Inverse or Pseudo-Inverse method.
- **Jacobian Matrix**: This matrix relates end effector velocities to joint velocities. It's used in iterative methods to adjust joint angles and minimize errors between desired and actual positions.
- **Singularities**: Singularities occur when the robot loses degrees of freedom, leading to undefined solutions. Handling singularities is crucial for robust IK solutions.
- **IK Solver in MoveIt**: MoveIt provides robust IK solvers that can handle complex robotic configurations. These solvers integrate with ROS to compute the necessary joint angles for desired end effector positions efficiently. MoveIt supports several IK solvers, such as KDL, IKFast, and TRAC-IK, offering flexibility and precision in planning and executing robot movements.

Procedure:

1. Go to `https://github.com/xArm-Developer/xarm_ros` and download the configuration files.
2. Create a new package with the above files in a workspace. Suppose you have ‘catkin_ws‘ as your workspace, then go to ‘catkin_ws¿¿src‘ and copy the above extracted data.
3. Open the MoveIt configuration of xarm5 using the following command:

   ```
   roslaunch xarm5_moveit_config demo.launch
   ```

4. Plan the architecture using MoveIt, by moving the markers, then click on plan and execute as per the following figure. (insert figure)



Robot Description: robot_description
Planning Scene Topic: move_group/monito...
Planning Group: xarm5
Start State: <current>
Goal State: <current>
Path Constraints: None
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 1.00
Accel. Scaling: 1.00
☐ Use Cartesian Path
☑ Collision-aware IK
☐ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning

Figure 1: Planning and Executing in MoveIt

5. Get the end effector coordinates using the following Python code in your ROS package:

   ```
   #!/usr/bin/env python3
   import rospy
   import tf

   if __name__ == '__main__':
   rospy.init_node('tf_listener')
   listener = tf.TransformListener()

   rate = rospy.Rate(10.0)
   while not rospy.is_shutdown():
   try:
   (trans, rot) = listener.lookupTransform('/link_base', '/link5', rospy.Time(0))
   # trans contains the translation vector (x, y, z)
   # rot contains the rotation quaternion (x, y, z, w)
   rospy.loginfo("Transform from frame1 to frame2: translation={}, rotation={}".format(trans, rot)
   except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
   continue

   rate.sleep()
   ```

6. Get the joint states by running the code:

   ```
   rostopic echo /jointstates
   ```

7. Apply the joint values in XARM5 and test the end effector coordinate values. Complete the table:
8. Compare the values of both and compile the results.

End effector coordinates	J1	J2	J3	J4	J5
MoveIt config value					
Real value					

Table 1: Comparison of MoveIt and Real Values

Results:

Experiment No: 5
Mobile Robot Moving to a point and following a line

Objective

- To program a mobile robot to move to a point.
- To program a mobile robot following a straight line.

Theory

Mobile Robots

A mobile robot is a machine controlled by software that uses sensors and other technology to identify its surroundings and move around its environment. Mobile robots function using a combination of artificial intelligence (AI) and physical robotic elements, such as wheels, tracks, and legs. Mobile robots are becoming increasingly popular across different business sectors. They are used to assist with work processes and even accomplish tasks that are impossible or dangerous for human workers.

Unicycle Model

A unicycle model of controlling a mobile robot is a simplified modeling approach modified from the differential drive mobile robots.

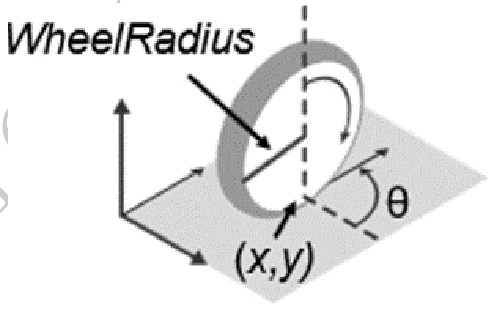


Figure 1: Unicycle Structure

$$\dot{x} = v\cos(\theta)$$
$$\dot{y} = v\sin(\theta)$$
$$\dot{\theta} = \omega$$

Where v and ω are the linear and angular velocities of the robot.

Moving to a Point

Consider the problem of moving toward a goal point (x^*, y^*) in the plane. The linear velocity is given by

$$v = k\sqrt{(x^* - x)^2 + (y^* - y)^2}$$

and to steer toward the goal which is at the vehicle-relative angle in the world frame with angular velocity

$$\omega = k_\omega(\theta^* - \theta)$$

Following a Line

A mobile robot is to follow a line on the plane defined by $ax + by + c = 0$. This requires two controllers to adjust the angular velocity with the linear velocity kept constant. One controller steers the robot to minimize the robot's normal distance from the line which according to the equation

$$d = \frac{(a, b, c) \cdot (x, y, 1)}{\sqrt{a^2 + b^2}}$$

The proportional controller

$$\alpha_d = -K_d d, \quad K_d > 0$$

turns the robot toward the line. The second controller adjusts the heading angle, or orientation, of the vehicle to be parallel to the line

$$\theta^* = -\frac{a}{b}$$

using the proportional controller

$$\alpha_h = K_h(\theta^* - \theta), \quad K_h > 0$$

The combined control law

$$\gamma = -K_d d + K_h(\theta^* - \theta)$$

turns the steering wheel so as to drive the robot toward the line and move along it.

Procedure

1. Create model
2. Open MATLAB and Simulink.
3. Create a unicycle model as per the diagram.

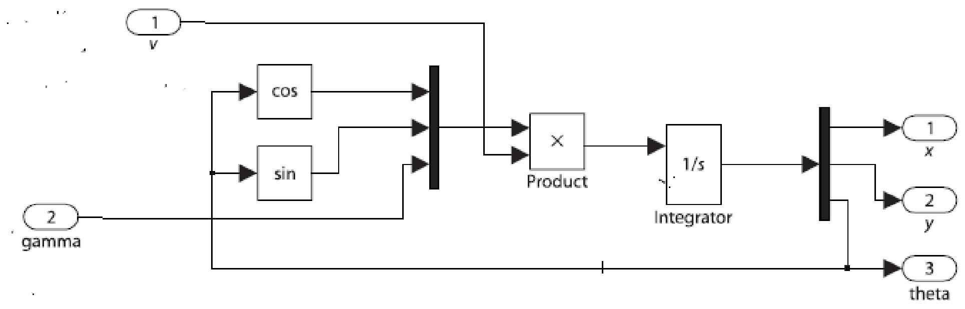


Figure 2: Simulink model for unicycle model

```
>> xg = [5 5];
```

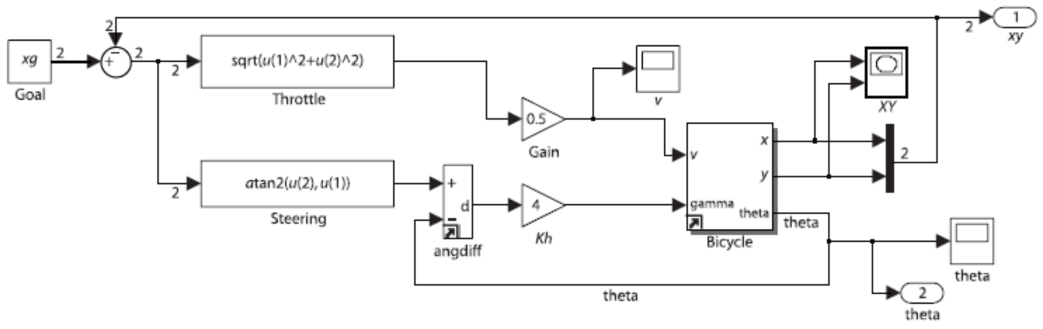


Figure 3: Simulink model for Moving to Point

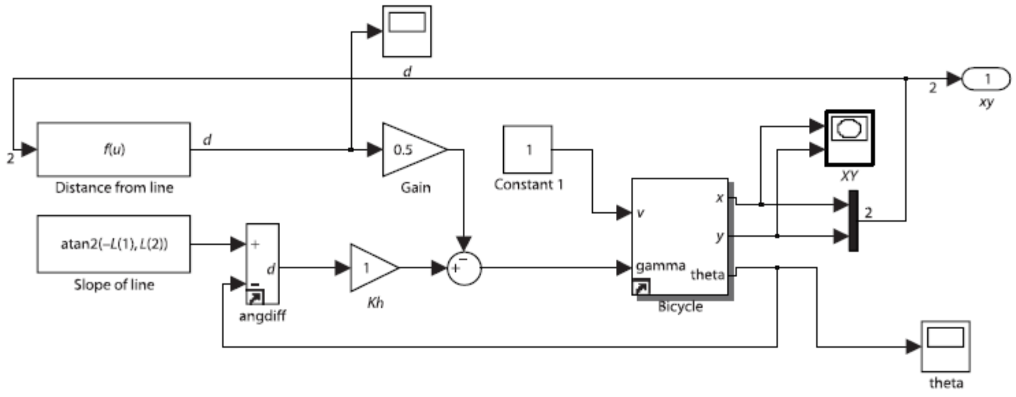


Figure 4: Simulink model for Following a Line

4. The initial position of the robot is given in integrator as a workspace variable `x0`; which is a 1×3 vector given by [0 0 0];

5. Save the unicycle model (Refer calculation) as a subsystem. (select all ¿¿ right click ¿¿ create subsystem.)

6. Moving to a point
   (a) Define the change in initial position if any by changing the variable `x0`
   (b) Define the final position `xg` in workspace.
   (c) Complete the block diagram “Moving to a point” and run the same.

7. Following a line
   (a) Specify the target line as a 3-vector (a, b, c) as a variable `L`. For example, if (a, b, c) are (1, -2, 4) then `L = [1 -2 4]`
   (b) Define the initial pose in the integrator by equating the workspace variable, for example: `x0 = [8 5 pi/2]`
   (c) Complete the block diagram for following a line in Simulink and run the code.

Results:

Experiment No: 6
Obstacle Avoidance of Mobile Robot

Objective

- Obstacle avoidance while the robot follows a path guided by a line.

Theory

Obstacle avoidance is a critical functionality in mobile robotics, ensuring that the robot can navigate safely and efficiently in dynamic environments. The objective is to design a control system that enables the robot to follow a predefined path while detecting and avoiding obstacles in its trajectory.

In this experiment, Simulink software is employed to design the control logic. The robot uses an ultrasonic sensor to measure the distance to obstacles in its path. When an obstacle is detected within a certain range, the control system adjusts the robot's path to avoid collision.

The steps for obstacle avoidance involve:

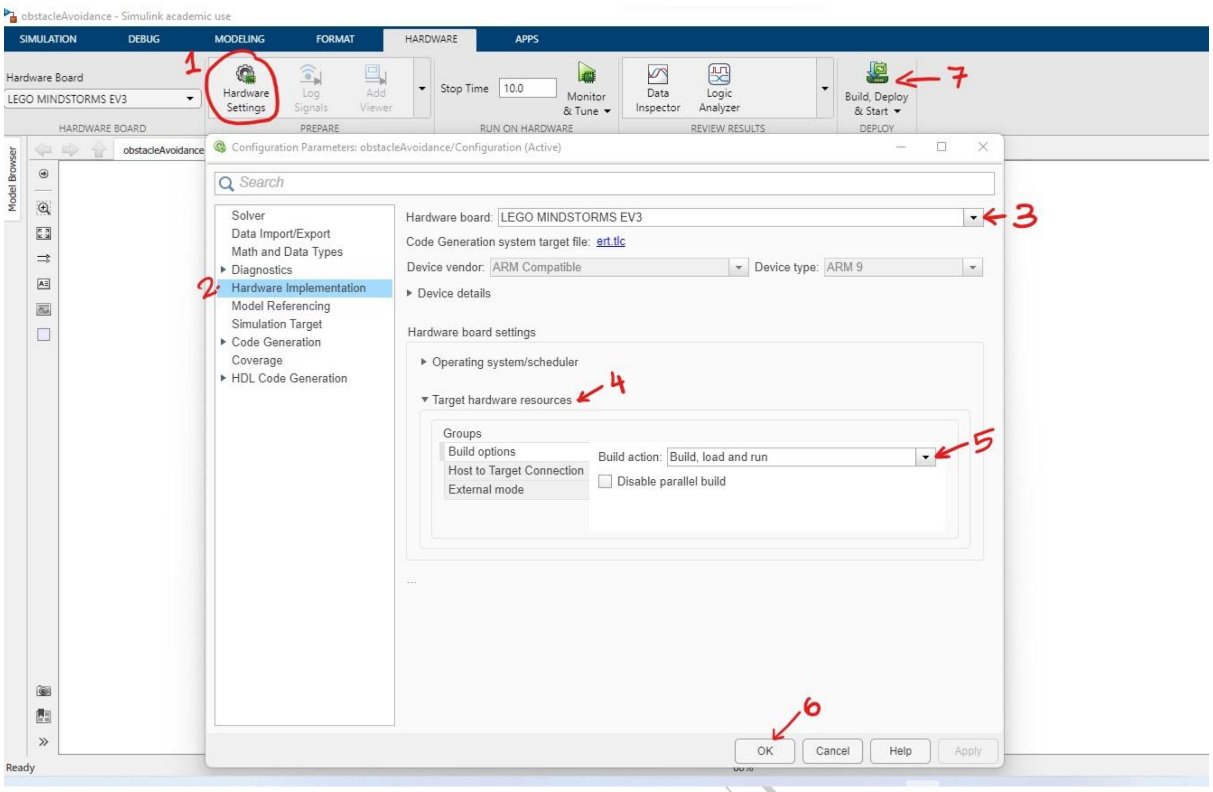
1. Detecting the obstacle using an ultrasonic sensor.
2. Calculating the distance to the obstacle.
3. Adjusting the robot's trajectory based on the distance information.

The control logic is implemented using Simulink, which provides a visual environment for modeling, simulating, and analyzing dynamic systems. The "Lego EV3 Mindstorms" library in Simulink contains the necessary components to build the model.

The ultrasonic sensor continuously monitors the distance to any obstacles. When an obstacle is detected within a predefined threshold, the control logic modifies the robot's movement to steer around the obstacle and return to the original path after passing the obstacle.

Procedure

1. Open Matlab software.
2. Open the Simulink tab.
3. Take necessary components from the "Lego EV3 Mindstorms" library.
4. Prepare the Simulink model for obstacle avoidance.
5. Upload the software to the robot by selecting the "Hardware" tab and sequentially following the numbered steps in the following diagram (insert figure).



Hardware Board: LEGO MINDSTORMS EV3

Stop Time: 10.0

Hardware board: LEGO MINDSTORMS EV3

Code Generation system target file: ert.tlc

Device vendor: ARM Compatible

Device type: ARM 9

Build action: Build, load and run

Figure 1: Simulink Interface

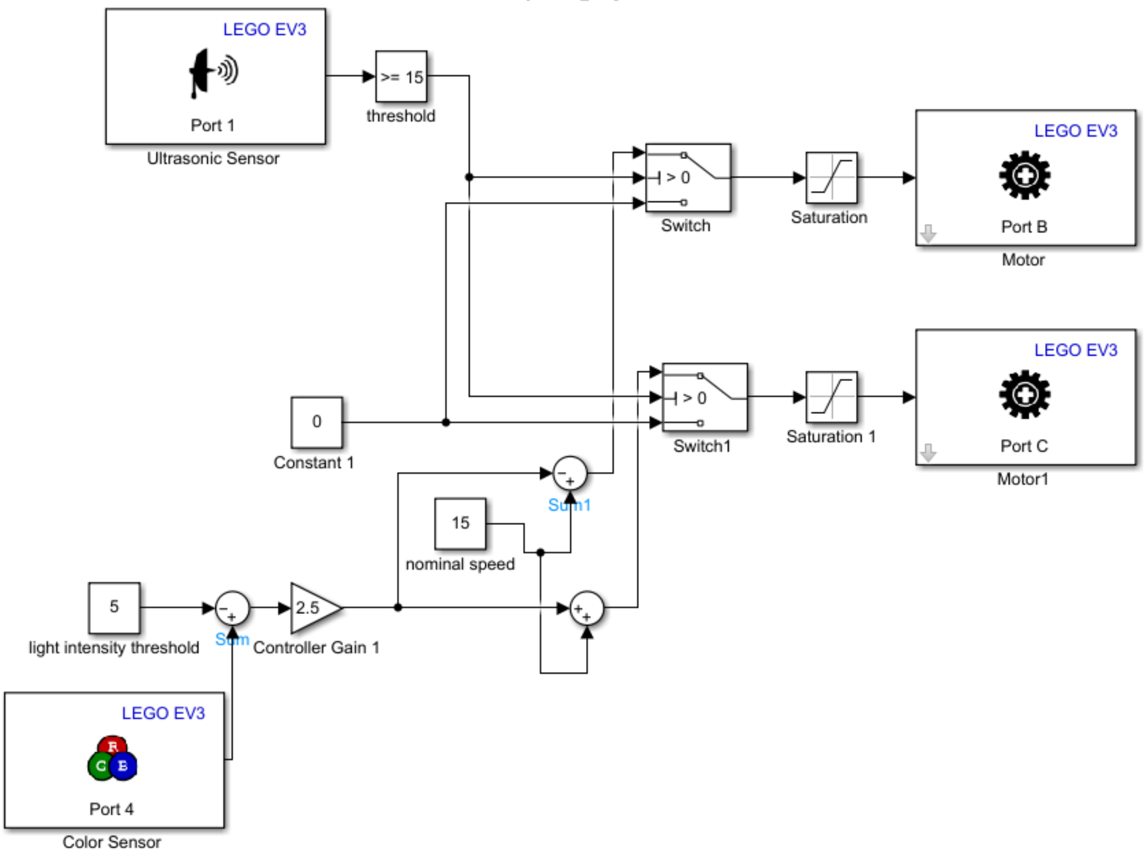


Figure 2: Block Diagram

Results:

Experiment No: 7
Localization of Mobile Robot using LiDAR

Objective

Understand how a mobile robot can perform localization. Perform SLAM (Simultaneous Localization and Mapping) operation, save the map, and navigate through the map.

Theory

Localization refers to the process of a robot identifying its position in the environment. The SLAM operation is performed using a LIDAR in the turtlebot to understand localization. In this experiment, map creation is also performed along with localization.

LIDAR (Light Detection and Ranging) is a remote sensing method that uses light in the form of a pulsed laser to measure ranges (variable distances) to obstacles. These light pulses generate precise, three-dimensional information about the shape of the obstacle and its surface characteristics.

Procedure

Procedure for SLAM

1. Open a terminal in Ubuntu

```
>> cd catkin_ws
>> TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

2. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

3. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

4. Now move the turtlebot in the world using control keys on the keyboard allotted by teleop. Once the entire area is scanned, proceed to save the map by following the steps below.

5. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> rosrun map_server map_saver -f ~/map
```

Procedure for Navigation

1. Terminate all applications with Ctrl + C that were launched in the previous sections. Type in a new terminal

```
>> cd catkin_ws
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

2. Run the Navigation Node:
3. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

4. Estimate Initial Pose: Initial Pose Estimation must be performed before running the Navigation. TurtleBot3 has to be correctly located on the map with the LDS sensor data that neatly overlaps the displayed map.
5. Click the 2D Pose Estimate button in the RViz menu.
6. Click on the map where the actual robot is located and drag the large green arrow toward the direction where the robot is facing.
7. Repeat steps 1 and 2 until the LDS sensor data is overlaid on the saved map.
8. Launch the keyboard teleoperation node to precisely locate the robot on the map.

```
>> roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

9. Move the robot back and forth a bit to collect the surrounding environment information and narrow down the estimated location of the TurtleBot3 on the map which is displayed with tiny green arrows. Terminate the keyboard teleoperation node by entering Ctrl + C to the teleop node terminal in order to prevent different cmd_vel values from being published from multiple nodes during Navigation.

Result

Experiment No: 7
Localization of Mobile Robot using LiDAR

Objective

Understand how a mobile robot can perform localization. Perform SLAM (Simultaneous Localization and Mapping) operation, save the map, and navigate through the map.

Theory

Localization refers to the process of a robot identifying its position in the environment. The SLAM operation is performed using a LIDAR in the turtlebot to understand localization. In this experiment, map creation is also performed along with localization.

LIDAR (Light Detection and Ranging) is a remote sensing method that uses light in the form of a pulsed laser to measure ranges (variable distances) to obstacles. These light pulses generate precise, three-dimensional information about the shape of the obstacle and its surface characteristics.

Procedure

Procedure for SLAM

1. Open a terminal in Ubuntu

```
>> cd catkin_ws
>> TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

2. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

3. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

4. Now move the turtlebot in the world using control keys on the keyboard allotted by teleop. Once the entire area is scanned, proceed to save the map by following the steps below.

5. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> rosrun map_server map_saver -f ~/map
```

Procedure for Navigation

1. Terminate all applications with Ctrl + C that were launched in the previous sections. Type in a new terminal

```
>> cd catkin_ws
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

2. Run the Navigation Node:
3. Open a new terminal window at `catkin_ws`

```
>> export TURTLEBOT3_MODEL=burger
>> roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

4. Estimate Initial Pose: Initial Pose Estimation must be performed before running the Navigation. TurtleBot3 has to be correctly located on the map with the LDS sensor data that neatly overlaps the displayed map.
5. Click the 2D Pose Estimate button in the RViz menu.
6. Click on the map where the actual robot is located and drag the large green arrow toward the direction where the robot is facing.
7. Repeat steps 1 and 2 until the LDS sensor data is overlaid on the saved map.
8. Launch the keyboard teleoperation node to precisely locate the robot on the map.

```
>> roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

9. Move the robot back and forth a bit to collect the surrounding environment information and narrow down the estimated location of the TurtleBot3 on the map which is displayed with tiny green arrows. Terminate the keyboard teleoperation node by entering Ctrl + C to the teleop node terminal in order to prevent different cmd_vel values from being published from multiple nodes during Navigation.

Result