WAREHOUSE AUTOMATION USING ROS BASED ROBOTIC ARM FOR PICK AND PLACE TASKS

A Course Project Report

Submitted to the APJ Abdul Kalam Technological University

in partial fulfillment of requirements for the award of degree

Master of Technology

in

Robotics and Automation

by

MAHESH MOHANDAS(TVE24ECRA06)

POOJA GOPAN(TVE24ECRA10)

SREERAG SS(TVE24ECRA15)

Robotics Lab

221 LIA 001

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING TRIVANDRUM

KERALA

JANUARY 2025

DEPT. OF ELECTRONICS & COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING TRIVANDRUM

2024 - 25

CERTIFICATE

This is to certify that the report entitled **WAREHOUSE AUTOMATION USING A ROS BASED ROBOTIC ARM FOR PICK AND PLACE TASKS** submitted by **MAHESH MOHANDAS** (TVE24ECRA06), **POOJA GOPAN** (TVE24ECRA10) **SREERAG S S** (TVE24ECRA15) to the APJ Abdul Kalam Technological University in partial fulfillment of the M.Tech. degree in Robotics and Automation is a bonafide record of the course project work carried out by him under our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Prof. Joaquim Ignatious Monteiro

Assistant Professor

Department of Electronics and Communication Engineering

College of Engineering Trivandrum

Thiruvananthapuram, Kerala

Abstract

This project focuses on designing and implementing a robotic arm controlled by the Robot Operating System (ROS) to perform autonomous pick-and-place tasks within a warehouse environment. The system aims to minimize reliance on manual labor while significantly improving the speed, efficiency, and precision of order fulfillment processes. By integrating advanced capabilities such as target detection using infrared sensors, ROS-based path planning, and real-time control and coordination, the robotic arm can effectively identify, locate, and manipulate objects within a warehouse setting. Leveraging ROS 1 (Noetic), the project incorporates modularity, sensor integration, and simulation to create a unified framework for handling complex robotic operations, including movement planning and object perception. This system not only showcases the potential of ROS in transforming logistics but also serves as a scalable and cost-effective model for warehouse automation, capable of performing multi-utility tasks such as sorting, packing, and order consolidation. By streamlining workflows and enhancing operational accuracy, this project highlights the transformative impact of ROS-based robotic systems in modernizing the logistics industry.

Contents

Abstract **i**

List of Figures **iv**

1 Introduction **1**

2 Design **3**

2.1 METHODOLOGY 3

2.2 DESIGN TOOLS 7

3 Results **9**

3.1 SCREENSHOTS OF SIMULATION 12

3.1.1 ZERO PAUSE 12

3.1.2 STRAIGHT UP 12

3.1.3 PICK OBJECT POSE 12

3.1.4 LIFT OBJECT POSE 13

3.1.5 OPPOSITE POSE 13

3.1.6 PLACE OBJECT OPPOSITE POSE 13

4 Discussion **14**

4.1 INTEGRATION WITH ADVANCED VISION SYSTEMS 14

4.2 DYNAMIC OBSTACLE AVOIDANCE 14

4.3 MULTI ROBOT COORDINATION 15

4.4 IoT INTEGRATION FOR WAREHOUSE MANAGEMENT 15

4.5 ENHANCED MOBILITY 15

4.6 IMPLEMENTATION OF MACHINE LEARNING AND AI 16

4.7 ENERGY EFFICIENCY AND SUSTAINABILITY 16
4.8 SCALABILITY TO LARGE SWAREHOUSES 16

5 Code **17**

References **25**

List of Figures

1.1 5 DOF ROBOTIC ARM 2

3.1 5-DOF ROBOTIC ARM PICK UP 11
3.2 5-DOF ROBOTIC ARM PLACE 11
3.3 ZERO PAUSE 12
3.4 STRAIGHT UP 12
3.5 PICK OBJECT POSE 12
3.6 LIFT OBJECT POSE 13
3.7 OPPOSITE POSE 13
3.8 PLACE OBJECT OPPOSITE POSE 13

Chapter 1

Introduction

A robotic arm with 5 degrees of freedom (DOF) is a sophisticated manipulator designed for precise and versatile tasks in dynamic environments, such as pick-and-place operations in warehouse automation. The 5-link configuration utilizes revolute joints, which allow rotational motion around fixed axes, enabling a high degree of control over the arm's angular positioning. The base joint facilitates horizontal rotation, providing azimuthal movement that allows the arm to pivot and reach objects in different directions. The shoulder and elbow joints control vertical movement and adjust the arm's reach, making it capable of accessing objects at varying heights and depths. The final two revolute joints in the wrist enable pitch and roll motions, ensuring precise orientation and alignment of the end-effector, which is critical for manipulating objects or interacting with other systems.

In more advanced configurations, the robotic arm can be extended to include 6 or 7 links by incorporating prismatic joints. Prismatic joints allow linear motion along a specified axis, adding the capability to extend or contract the arm's length. A 6-link configuration might include one prismatic joint, enabling the arm to extend horizontally to reach farther objects without requiring significant angular adjustments. In a 7-link setup, two prismatic joints could be integrated—one for horizontal extension and another for vertical adjustment—providing even greater flexibility and adaptability in environments with spatial constraints or diverse object placements.

This combination of revolute and prismatic joints significantly enhances the robotic arm's functionality. The revolute joints provide the rotational freedom necessary for maneuvering around obstacles and accurately positioning the end-effector, while the prismatic joints expand the workspace and allow fine-tuned adjustments for handling objects at various distances and heights. Such a design is particularly advantageous in warehouse settings, where tasks often require interacting with objects stored at different locations and orientations. The robotic arm's ability to combine precise rotational and linear movements ensures high efficiency, accuracy, and adaptability, making it an indispensable tool for modern automation applications.

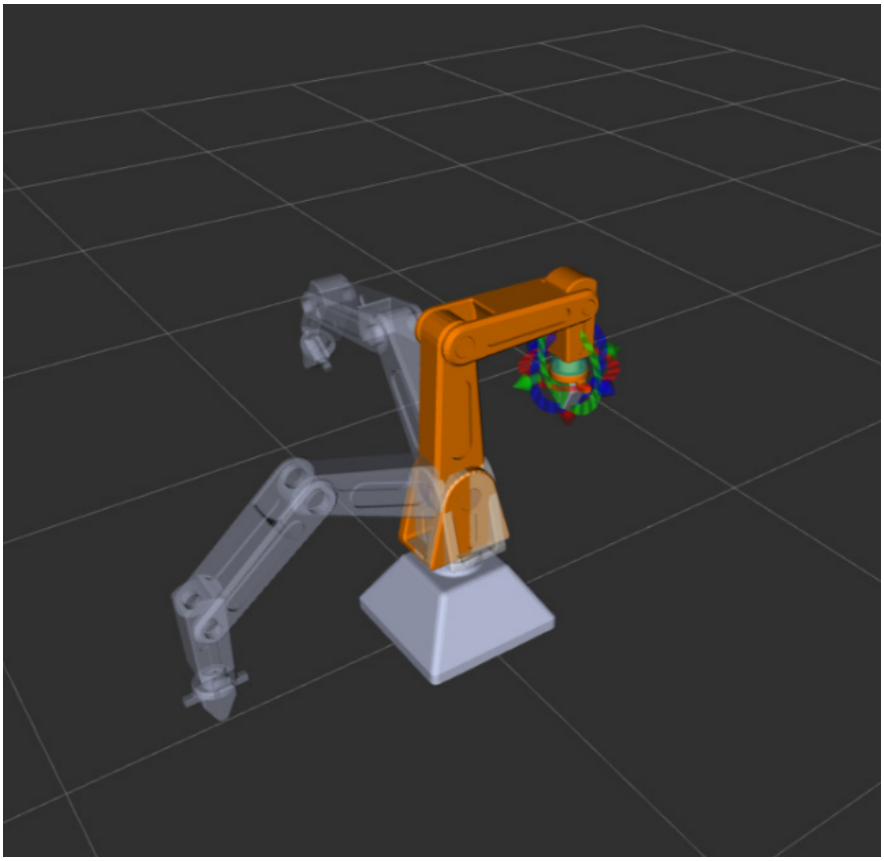

Figure 1.1: 5 DOF ROBOTIC ARM

Chapter 2

Design

The 5-DOF robotic arm simulation is designed using Gazebo for physics modeling and RViz for visual verification of joint movements. Key poses, including zero pause, straight up, pick object, lift object, opposite pose, and place object, are defined, and joint angles are computed in radians and published using ROS. These angles are converted to degrees and transmitted to an Arduino, which controls six servo motors (three MG995 high-torque and three SG90 micro servos) to perform the arm's movements. The robotic arm executes pick-and-place operations, with all motions tested and fine-tuned in simulation before physical implementation.

The 5-DOF robotic arm simulation is designed using ROS, Gazebo, RViz, and MoveIt to enable realistic motion planning and visualization. The arm is modeled in URDF/XACRO and simulated in Gazebo with RViz providing real-time visualization of joint states. MoveIt handles motion planning and collision detection, enabling smooth, collision-free pick-and-place operations.

2.1 METHODOLOGY

How to prepare a Technical report using LaTeXMethodology Report: 5-DOF Robotic Arm Simulation and Integration with ROS and Arduino

1. Simulation Setup

Gazebo and RViz

Gazebo

- Create a 5-DOF robotic arm model in URDF or SDF format.
  - Define joints, links, and actuators using appropriate joint types (revolute, continuous, or prismatic).
  - Include physical properties such as mass, inertia, and collision parameters for each link.
  - Attach a gripper to the end-effector to enable pick-and-place functionality.
- Configure the Gazebo simulation environment to include:
  - A flat surface or table for placing objects.
  - Target objects for the robotic arm to manipulate.

RViz

- Use the same URDF model to load the robotic arm in RViz.
- Add the following components for visualization and interaction:
  - Joint state publishers and robot state publishers to display real-time joint movements.
  - Interactive markers to manually control the arm's pose.

Launch Files

- Develop ROS launch files for:
  - Initializing the Gazebo environment with the robotic arm.
  - Loading the arm model in RViz for visualization.
  - Including necessary plugins and controllers, such as the `gazebo_ros_control` plugin for joint control.

2. ROS Nodes

Joint State Publisher

- Create a node to publish joint angles (in radians) for all six joints, including the gripper.
- Define and implement the following predefined poses:
  - **Zero Pose**: All joint angles set to zero.
  - **Straight-Up Pose**: The arm is extended vertically upward.
  - **Pick Object Pose**: The arm is positioned to grasp an object.
  - **Lift Object Pose**: The arm lifts the object to a specified height.
  - **Opposite Pose**: The arm faces the opposite direction.
  - **Place Object Pose**: The arm moves to place the object in a specific location.

Joint Trajectory Controller

- Integrate a joint trajectory controller (e.g., `position_controllers/JointTrajectoryControl` to enable smooth, continuous motion between poses.
- Configure the controller using a ROS control YAML configuration file, specifying joint names and parameters.

Conversion Node

- Develop a ROS node to convert joint angles from radians (output by the simulation) to degrees.
- Publish the converted angles to a dedicated topic (e.g., `/joint_angles`), which will be subscribed to by the Arduino.

3. Communication with Arduino

Publishing Joint Angles

- Use the conversion node to publish joint angles in degrees to the topic `/joint_angles`.
- Ensure angles are updated in real-time as the arm moves in Gazebo.

Arduino Subscriber

- Implement a ROS node on Arduino to subscribe to the `/joint_angles` topic.
- Parse the received joint angle data and map it to control signals for the servo motors.

4. Arduino Integration

Hardware Control

- Use six servo motors for the robotic arm:
  - **MG995 (High-Torque Motors)**: Control heavier joints, such as the base and shoulder.
  - **SG90 (Micro Servos)**: Operate lighter joints, such as the wrist and gripper.

Control Logic

- Parse the joint angles received from the ROS topic.
- Map the angles to PWM signals suitable for controlling the servo motors.
- Implement interpolation techniques for smooth, gradual movement of the joints.

5. Testing and Tuning

Simulation Testing

- Verify the robotic arm's poses and movements in Gazebo.

- Validate joint state transitions using RViz visualization.

System Testing

- Test the complete system by sending predefined poses to the arm and observing its behavior.
- Fine-tune joint angles for precise pick-and-place operations.
- Debug communication between ROS and Arduino to ensure data integrity and synchronization.

Physical Testing

- Deploy the system on a physical robotic arm and test real-world performance.
- Adjust motor control parameters, such as speed and torque limits, for optimal operation.

This methodology ensures the successful integration of a 5-DOF robotic arm simulation with ROS and Arduino, enabling efficient and accurate pick-and-place tasks in both simulated and real-world environments.

2.2 DESIGN TOOLS

Gazebo, RViz, and MoveIt are integral tools in the design and simulation of robotic systems, offering specialized capabilities that complement each other for effective development and testing. **Gazebo**is a physics-based simulation environment that accurately replicates real-world dynamics, such as gravity, friction, and collisions, allowing developers to test robotic systems in realistic conditions. It supports detailed modeling of robots using URDF or SDF files, which define the robot's structure, including joints, links, and actuators. Gazebo also provides sensor simulation and plugins for extending functionality, making it ideal for evaluating the robotic arm's movements, gripper functionality, and interactions with objects during tasks like pick-and-place.

RViz part of the ROS ecosystem, serves as a powerful visualization tool for monitoring and debugging robotic systems. It renders 3D models of robots, showing joint movements, sensor data, and motion trajectories in real-time. Developers can use RViz to visualize the robotic arm's state, validate its predefined poses, and interact with the system through features like interactive markers, enabling manual pose adjustments and fine-tuning of operations. By visualizing the arm's behavior alongside sensor outputs, RViz ensures precise and efficient debugging during simulation and integration.

MoveIt is a motion planning framework that integrates closely with ROS to enable advanced functionalities for controlling robotic movements. It provides tools for path planning, trajectory generation, and collision avoidance, ensuring smooth and efficient transitions between the robotic arm's poses. MoveIt also includes inverse kinematics solvers to calculate joint configurations for achieving desired end-effector positions. Additionally, its integration with RViz enables users to visualize planned trajectories, detect and resolve potential collisions, and interactively adjust motion plans. This makes MoveIt a crucial tool for ensuring the arm's movements are both precise and safe.

Together, **Gazebo, RViz, and MoveIt** form a robust toolchain for designing, simulating, and validating robotic systems. Gazebo handles the physical simulation, RViz provides real-time visualization and debugging, and MoveIt ensures effective motion planning and collision-free operation. These tools are indispensable for developing the 5-DOF robotic arm, enabling efficient testing and integration in both simulated and real-world environments.

Chapter 3

Results

To evaluate and validate the design and functionality of the robotic arm, simulations were conducted using Gazebo and RViz, two powerful tools integrated with the Robot Operating System (ROS). These simulations served as critical steps in developing, testing, and optimizing the robotic arm's performance before physical implementation.

Simulation in Gazebo

Gazebo is a robust physics-based simulation environment that allows realistic modeling of robots and their interactions with the environment. In this project, Gazebo was used to simulate the 5 DOF robotic arm, including its links, joints (both revolute and prismatic), sensors, and end-effector. A detailed URDF (Unified Robot Description Format) file was created to define the arm's geometry, including its links, revolute joints, and prismatic joints for extended configurations. Additionally, material properties, such as mass and inertia, were defined to simulate the arm's dynamics accurately.

Gazebo's physics engine enabled precise simulation of real-world forces, such as gravity, friction, and collisions. This allowed the robotic arm's behavior, including movement precision and joint torque, to be evaluated under realistic conditions. The virtual environment in Gazebo was populated with objects, enabling the testing of pick-and-place tasks. Motion planning algorithms ensured the robotic arm could execute these tasks smoothly and efficiently.

Furthermore, infrared sensors were integrated virtually in Gazebo to simulate

object detection. This validated the arm's ability to perceive its surroundings, locate objects accurately, and perform autonomous operations. The motion planning and trajectory generation were also simulated in Gazebo using ROS libraries like MoveIt, ensuring obstacle-free paths and precise task execution.

Visualization in RViz

RViz is a visualization tool used for real-time monitoring and debugging of robotic systems in ROS. While Gazebo handled the physics-based simulation, RViz provided a visual representation of the robot's state and interactions with the environment, offering valuable insights into its performance.

In RViz, the robotic arm's joints, links, and end-effector movements were displayed in real time. This included updates on joint angles, link positions, and the arm's orientation during task execution. Trajectories generated by motion planning libraries, such as those created using MoveIt, were visualized, ensuring they were collision-free and met task requirements.

RViz also displayed ROS's TF (transform) frames, which represented the position and orientation of the robotic arm and its components in the workspace. Infrared sensor data used for detecting objects in Gazebo was visualized in RViz, confirming successful object detection and proper communication between the sensors and the robotic arm. Additionally, RViz included a representation of the virtual warehouse environment, showing the objects to be picked and placed, obstacles, and workspace boundaries, making it easier to assess the arm's interaction with its surroundings.

Hardware Realization

The hardware for the 5-DOF robotic arm system is designed to ensure efficient and precise pick-and-place tasks, incorporating a robust structure, reliable actuation, and seamless control. The robotic arm comprises lightweight yet durable materials, with five degrees of freedom enabled by rotational joints for base, shoulder, elbow, wrist, and gripper movements. High-torque MG995 servo motors drive the heavier joints, while lightweight SG90 micro servos handle finer movements, such as wrist control and gripper operation. An Arduino microcontroller serves as the central control unit, interfacing with the ROS environment to receive joint angles and convert them into PWM signals for servo control. A stable power supply, typically a 6V–7.4V

battery pack with a voltage regulator, ensures consistent operation of all components. The end-effector is a two-finger parallel gripper actuated by an SG90 servo, offering precision and adaptability for handling objects of varying sizes. Sensors, including limit switches and optional position feedback devices, enhance control and prevent over-rotation. Communication between the Arduino and ROS is achieved through USB or wireless modules, ensuring real-time data exchange. Together, these components form a well-integrated hardware system that supports the robotic arm's performance in both simulated and real-world environments.

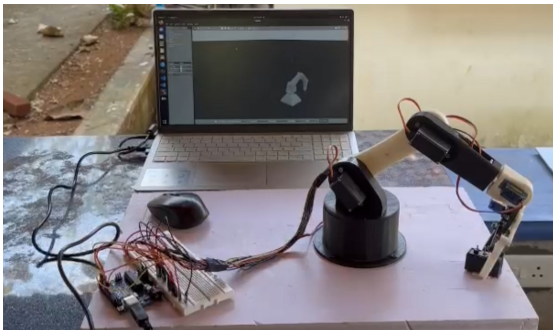

Figure 3.1: 5-DOF ROBOTIC ARM PICK UP

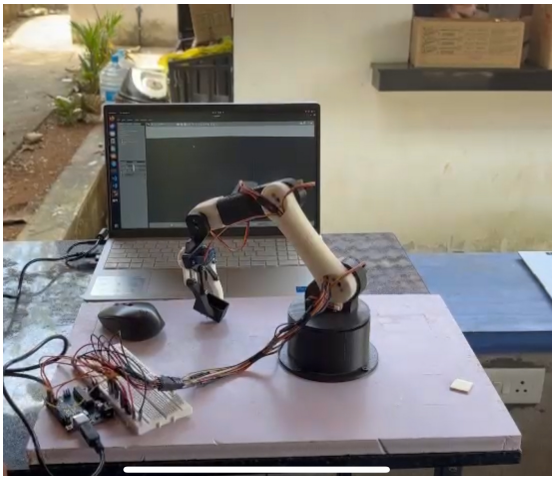

Figure 3.2: 5-DOF ROBOTIC ARM PLACE

3.1 SCREENSHOTS OF SIMULATION

3.1.1 ZERO PAUSE

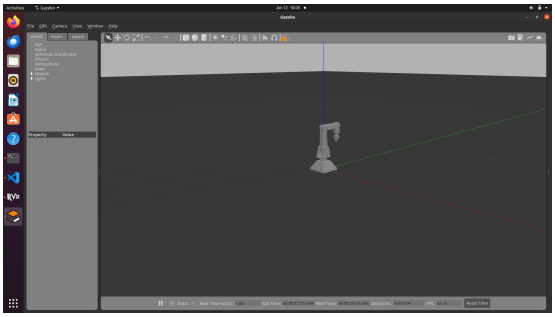

Figure 3.3: ZERO PAUSE

3.1.2 STRAIGHT UP

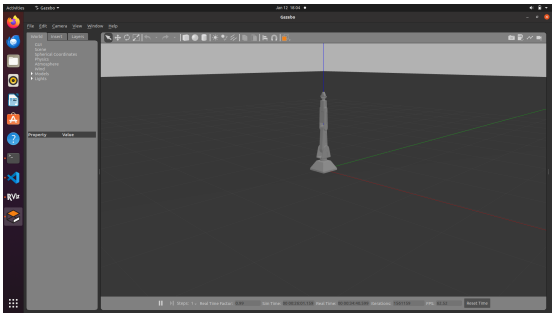

Figure 3.4: STRAIGHT UP

3.1.3 PICK OBJECT POSE

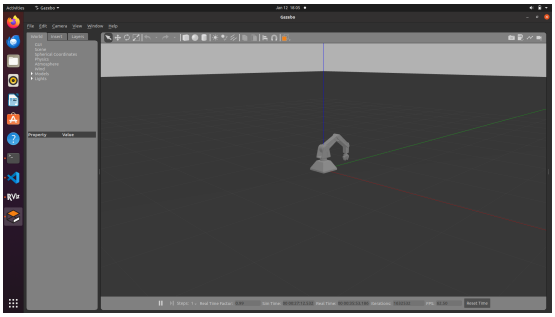

Figure 3.5: PICK OBJECT POSE

3.1.4 LIFT OBJECT POSE

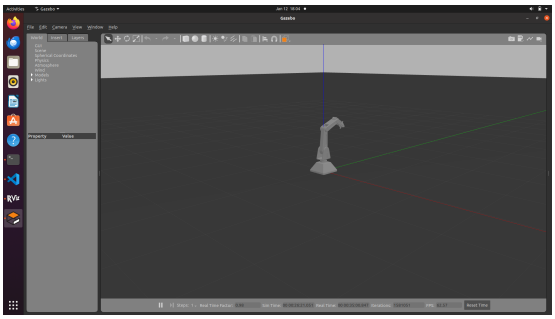

Figure 3.6: LIFT OBJECT POSE

3.1.5 OPPOSITE POSE

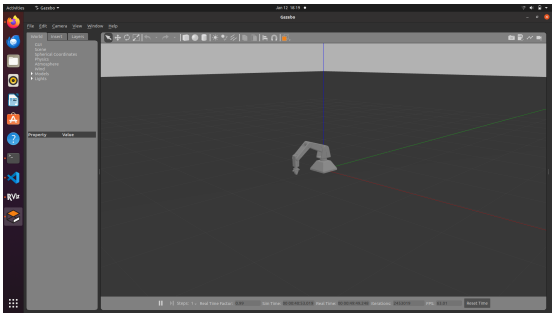

Figure 3.7: OPPOSITE POSE

3.1.6 PLACE OBJECT OPPOSITE POSE

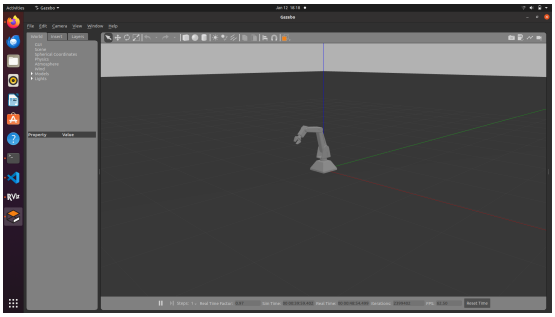

Figure 3.8: PLACE OBJECT OPPOSITE POSE

Chapter 4

Discussion

The integration of a ROS-based robotic arm in warehouse automation opens the door to numerous future extensions, further advancing operational efficiency, scalability, and adaptability. While the current implementation focuses on pick-and-place tasks, future developments could leverage emerging technologies and advanced ROS capabilities to enhance the system's functionality and broaden its applications.

4.1 INTEGRATION WITH ADVANCED VISION SYSTEMS

Future iterations could incorporate advanced machine vision systems using high-resolution cameras and AI-driven image recognition algorithms. This would enable the robotic arm to identify a wider variety of objects, even those with irregular shapes or diverse packaging. Using ROS packages like OpenCV and perception libraries, the arm could classify, sort, and handle items with higher precision.

4.2 DYNAMIC OBSTACLE AVOIDANCE

By integrating dynamic obstacle avoidance capabilities, the robotic arm could operate in real-time within ever-changing warehouse environments. Utilizing LiDAR, 3D mapping, or SLAM (Simultaneous Localization and Mapping), the arm could detect

and adapt to obstacles, ensuring uninterrupted operations. ROS packages such as Navigation Stack and MoveIt could be extended to handle real-time path re-planning.

4.3 MULTI ROBOT COORDINATION

Expanding the system to include multiple robotic arms working in coordination would significantly enhance throughput. ROS's robust multi-node communication system can facilitate real-time data exchange between robots, enabling task allocation, load balancing, and synchronized operations. A fleet of collaborative robots could automate large-scale sorting and order fulfillment processes seamlessly.

4.4 IoT INTEGRATION FOR WAREHOUSE MANAGEMENT

Integration with IoT-enabled warehouse management systems would allow the robotic arm to operate as part of a connected ecosystem. Sensors placed throughout the warehouse could provide real-time data on inventory levels, item locations, and task priorities. This integration would enable the robotic arm to retrieve items efficiently and assist in just-in-time inventory systems.

4.5 ENHANCED MOBILITY

The robotic arm could be mounted on an autonomous mobile robot (AMR) or an automated guided vehicle (AGV), allowing it to perform pick-and-place operations across different warehouse zones. ROS's Navigation Stack could be leveraged to guide the mobile base while maintaining precise control of the robotic arm, enabling tasks such as shelf restocking and dynamic order sorting.

4.6 IMPLEMENTATION OF MACHINE LEARNING AND AI

Machine learning models could be employed to optimize the robotic arm's operations over time. By analyzing task performance data, the system could learn to improve efficiency, predict maintenance needs, and adapt to new tasks without extensive reprogramming. ROS2, with its enhanced support for AI and data handling, could be adopted for these capabilities.

4.7 ENERGY EFFICIENCY AND SUSTAINABILITY

Future extensions could focus on energy-efficient operations, incorporating smart power management systems. Solar-powered or low-energy robotic arms could reduce operational costs and align with sustainable warehouse practices.

4.8 SCALABILITY TO LARGE SWAREHOUSES

By combining the robotic arm's pick-and-place capabilities with warehouse automation systems like conveyor belts, palletizers, and automated storage and retrieval systems (ASRS), the system could scale up for larger warehouses and distribution centers. ROS would serve as the backbone for seamless coordination between these components.

Chapter 5

Code

LAUNCH FILE

```
<launch>

            <include file = "$(find robot_arm_urdf)/
            launch/arm_urdf.launch" />


            <include file = "$(find movit_robot_arm_sim)/
            launch/move_group.launch" />


            <arg name="use_rviz" default="true" />
            <include file="$(find movit_robot_arm_sim)/
            launch/moveit_rviz.launch"
             if="$(arg use_rviz)">
                        <arg name="rviz_config
                        " value="$(find movit_robot_arm_sim)/
                        launch/moveit.rviz"/>
            </include>
</launch>
```

SIMULATION

```
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import actionlib
from math import pi

class MyRobot:


    def __init__(self, Group_Name):

        print(f"Initializing robot with group: {Group_Name}")
        self._commander = moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node('node_set_redefined_pose', anonymous=True)


        self._robot = moveit_commander.RobotCommander()

        self._scene = moveit_commander.PlanningSceneInterface()


        self._planning_group = Group_Name

        self._group = moveit_commander.MoveGroupCommander(self._planning_group)


        self._display_trajectory_publisher =
```

```
        rospy.Publisher('/move_group/display_planned_path',
                        moveit_msgs.msg.DisplayTrajectory, queue_size=1)


        self._exectute_trajectory_client =
        actionlib.SimpleActionClient('execute_trajectory',
                                     moveit_msgs.msg.ExecuteTrajectoryAction)
        self._exectute_trajectory_client.wait_for_server()


        self._planning_frame = self._group.get_planning_frame()
        self._eef_link = self._group.get_end_effector_link()
        self._group_names = self._robot.get_group_names()


        rospy.loginfo('\033[95m' +
                      "Planning Group: {}".format(self._planning_frame) + '\033[0m')
        rospy.loginfo('\033[95m' +
                      "End Effector Link: {}".format(self._eef_link) + '\033[0m')
        rospy.loginfo('\033[95m' +
                      "Group Names: {}".format(self._group_names) + '\033[0m')
        rospy.loginfo('\033[95m' + " >>> MyRobot initialization is done." + '\033[0m')

    def set_pose(self, arg_pose_name):
        rospy.loginfo('\033[32m'
                      + "Going to Pose: {}".format(arg_pose_name) + '\033[0m')


        self._group.set_named_target(arg_pose_name)


        success, plan, x, y = self._group.plan()
```

```
        if not success:
            rospy.logerr("Planning failed for pose: {}".format(arg_pose_name))
            return

        goal = moveit_msgs.msg.ExecuteTrajectoryGoal()

        goal.trajectory = plan

        self._exectute_trajectory_client.send_goal(goal)
        self._exectute_trajectory_client.wait_for_result()

        rospy.loginfo('\033[32m' + "Now at Pose: {}".format(arg_pose_name) + '\033[0m')

    def __del__(self):

        moveit_commander.roscpp_shutdown()
        rospy.loginfo('\033[95m' + "Object of class MyRobot Deleted." + '\033[0m')

def main():

    arm = MyRobot("arm_group")
    hand = MyRobot("hand")

    while not rospy.is_shutdown():

        arm.set_pose("zero_pose")
```

```
rospy.sleep(2)

arm.set_pose("straight_up")
rospy.sleep(2)


hand.set_pose("hand_open")
rospy.sleep(1)

arm.set_pose("pick_object_pose")
rospy.sleep(2)

hand.set_pose("hand_closed")
rospy.sleep(1)

arm.set_pose("lift_object_pose")
rospy.sleep(2)

arm.set_pose("place_object_opposit_pose")
rospy.sleep(2)


hand.set_pose("hand_open")
rospy.sleep(1)

arm.set_pose("opposite_pose")
rospy.sleep(2)


hand.set_pose("hand_closed")
rospy.sleep(1)
```

```
    del arm
    del hand



if __name__== '__main__':
    main()
```

PUBLISHER

```
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import JointState
import math

pub = rospy.Publisher('/joint_positions_string', String, queue_size=10)

def joint_state_callback(msg):
    # Convert position from radians to degrees and round to the nearest integer
    degrees_positions =
    [str(int(round(pos * (180 / math.pi)))) for pos in msg.position]
    degrees_positions[2]=
    str(-1* int(degrees_positions[2]))

    # Join the list of integer degree values into a single string separated by commas
    degrees_positions_str = ",".join(degrees_positions)

    rospy.loginfo("Joined Joint Positions in Degrees (Integer):
     %s", degrees_positions_str)

    # Create and publish the String message
    new_msg = String()
    new_msg.data = degrees_positions_str
    pub.publish(new_msg)
```

```
def listener():
    rospy.init_node('joint_state_listener', anonymous=True)
    rospy.Subscriber('/joint_states', JointState, joint_state_callback)
    rospy.spin()

if __name__ == '__main__':
    try:
        listener()
    except rospy.ROSInterruptException:
        pass
```

AURDINO CODE

```
#include <ros.h>
#include <std_msgs/String.h>
#include <Servo.h>

Servo s1,s2,s3,s4,s5,s6;

ros::NodeHandle nh;

int joint_positions[7]; // Declare an integer array to store the joint positions
int position_count = 0; // To keep track of how many positions we have

void jointStateCallback(const std_msgs::String& msg)
{


  String ss = msg.data;
  nh.loginfo(ss.c_str());

  int arr[7];
  int index = 0;
```

```
  int startIndex = 0;
  for (int i = 0; i < ss.length(); i++) {
    if (ss.charAt(i) == ',' || i == ss.length() - 1) {
      // Extract the number as a substring
      String numStr = ss.substring(startIndex, (i == ss.length() - 1) ? i + 1 : i);
      // Convert the substring to an integer and store it in the array
      arr[index++] = numStr.toInt();
      startIndex = i + 1;
    }

  }
  arr[5] = arr[5]*45;
  nh.loginfo(String(arr[5]).c_str());

  for (int i = 0; i < 7; i++) {
    nh.loginfo(String(arr[i]).c_str());
  }

  s1.write(arr[0]);
  s2.write(arr[1]);
  s3.write(arr[2]);
  s4.write(arr[3]);
  s5.write(arr[4]);
  s6.write(arr[5]);
}

ros::Subscriber<std_msgs::String> sub("/joint_positions_string", jointStateCallback);

void setup()
{
  Serial.begin(57600);
  nh.getHardware()->setBaud(57600);
  nh.initNode();
```

```
  nh.subscribe(sub);
  nh.loginfo("Subscribed to /joint_positions_string");

  s1.attach(2);
  s2.attach(4);
  s3.attach(7);
  s4.attach(8);
  s5.attach(9);
  s6.attach(11);
}

void loop()
{
  nh.spinOnce(); // Listen to incoming messages
  delay(10);   // Short delay to allow other processes
}
```

References

[1] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2015.

[2] S. Chitta, L. Marinho, V. Pradeep, and The ROS-Industrial Consortium, "ROS-Industrial: Open-Source Software for Manufacturing Automation," [Online]. Available: `https://rosindustrial.org`.

[3] N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sendai, Japan, 2004, pp. 2149–2154.

[4] R. Shah and P. Shah, "ROS-Based Autonomous Robotic Arm for Warehouse Automation," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 9, no. 6, pp. 41–48, Jun. 2020.

[5] I. Havoutis and S. Calinon, "Learning from Demonstrations and Motion Planning with ROS," *Frontiers in Robotics and AI*, vol. 6, no. 17, pp. 1–10, Feb. 2019.