UNIVERSITÉ DE TUNIS ELMANAR
ÉCOLE NATIONALE D'INGÉNIEURS DE TUNIS

Département Génie Électrique

Robotics Engineering

Robotic Arm Control using ROS 2 (Humble), Raspberry Pi & Arduino

Project Report

Prepared by:

Ahmed Mayekhav
Khalil Rezgui
Mohamed Yassin Ghomrasni
Wajdi Dridi

Supervised by:

Mme. Zohra Kardous
Mme. Nahla Khraief
Mme. Afef Hfaiedh

Academic Year: 2025/2026

Abstract

This report presents a comprehensive study and implementation of a 4 Degrees of Freedom (DOF) robotic arm control system. Beginning with an extensive literature review of robotics fundamentals, historical development, and contemporary applications, the document establishes the theoretical foundation necessary for understanding modern robotic manipulators. The practical implementation employs a distributed architecture utilizing ROS 2 Humble middleware on a Raspberry Pi for high-level control and trajectory planning, while an Arduino microcontroller handles low-level servo motor actuation. The system demonstrates effective integration of vision-based input processing, inverse kinematics computation, and real-time servo control through serial communication. This project showcases modern robotics development practices including containerization with Docker, modular ROS 2 node architecture, and hardware abstraction layers, bridging the gap between theoretical robotics concepts and practical implementation.

Contents

Part I
Theoretical Background and Literature Review

1 Introduction to Robotics

1.1 Definition and Scope

Robotics is an interdisciplinary field that integrates mechanical engineering, electrical engineering, computer science, and artificial intelligence to design, construct, and operate machines capable of performing tasks autonomously or semi-autonomously. A robot can be defined as a programmable, multi-functional manipulator designed to move materials, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks.

The field of robotics encompasses several key domains:

- **Industrial Robotics:** Manufacturing automation, assembly lines, welding, painting
- **Service Robotics:** Healthcare assistance, cleaning, delivery services
- **Mobile Robotics:** Autonomous vehicles, drones, exploration rovers
- **Collaborative Robotics:** Human-robot interaction, cobots in shared workspaces
- **Medical Robotics:** Surgical robots, rehabilitation devices, prosthetics

1.2 Historical Development

1.2.1 Early Concepts (Pre-1950s)

The concept of automated machines dates back to ancient civilizations, with early automata developed in Greece, China, and the Islamic Golden Age. However, the modern era of robotics began in the 20th century with the convergence of mechanical engineering and electronics.

1.2.2 The Birth of Industrial Robotics (1950s-1970s)

- **1954:** George Devol invented the first digitally operated programmable robot, called Unimate
- **1961:** Unimate was installed at General Motors for die casting and spot welding
- **1969:** Victor Scheinman developed the Stanford Arm, the first successful electrically powered, computer-controlled robot arm
- **1973:** KUKA introduced the first industrial robot with six electromechanically driven axes

1.2.3 Advancement Era (1980s-2000s)

This period saw exponential growth in robotics applications:

- Development of advanced sensors and vision systems
- Introduction of collaborative robots (cobots)
- Emergence of mobile robotics and autonomous navigation
- Integration of artificial intelligence and machine learning
- Miniaturization enabling medical and micro-robotics

1.2.4 Contemporary Robotics (2000s-Present)

Modern robotics is characterized by:

- Cloud robotics and distributed computing
- Deep learning for perception and control
- Soft robotics and biologically inspired designs
- Swarm robotics and multi-agent systems
- Human-robot collaboration in unstructured environments

2 Robotic Manipulators: Fundamental Concepts

2.1 Classification of Robotic Arms

2.1.1 By Mechanical Structure

Robotic manipulators can be classified based on their kinematic configuration:

1. **Cartesian (Gantry) Robots**
   - Three prismatic joints (PPP)
   - Rectangular workspace
   - High precision and repeatability
   - Applications: 3D printing, CNC machines, pick-and-place
2. **Cylindrical Robots**
   - One rotational and two prismatic joints (RPP)
   - Cylindrical workspace
   - Applications: Assembly operations, machine tending
3. **Spherical (Polar) Robots**
   - Two rotational and one prismatic joint (RRP)

- Spherical workspace
- Applications: Spot welding, material handling

4. **Articulated (Anthropomorphic) Robots**
   - Multiple rotational joints (RRR or more)
   - Complex, human-like workspace
   - Most versatile configuration
   - Applications: Welding, painting, assembly, our project
5. **SCARA (Selective Compliance Assembly Robot Arm)**
   - Two parallel rotational joints and one prismatic joint (RRP with vertical orientation)
   - Compliant in XY plane, rigid in Z direction
   - Applications: Assembly, packaging, PCB handling
6. **Delta (Parallel) Robots**
   - Parallel kinematic structure
   - Extremely high speed and precision
   - Applications: Pick-and-place, food packaging, pharmaceutical

2.1.2 By Degrees of Freedom

- **3-DOF:** Basic positioning (x, y, z)
- **4-DOF:** Positioning plus one orientation (our project)
- **5-DOF:** Positioning plus two orientations
- **6-DOF:** Full pose (position and orientation)
- **7+ DOF:** Redundant manipulators with additional flexibility

2.2 Kinematics Fundamentals

2.2.1 Forward Kinematics

Forward kinematics determines the position and orientation of the end-effector given the joint angles. For a serial manipulator, the transformation from base to end-effector is given by:

$$T_0^n = T_0^1 \cdot T_1^2 \cdot T_2^3 \cdots T_{n-1}^n \tag{1}$$

where T_i^{i+1} represents the homogeneous transformation matrix from frame i to frame $i+1$.

The Denavit-Hartenberg (DH) convention is commonly used to systematically describe the geometry of robot linkages. Each transformation is characterized by four parameters:

- a_i: link length
- α_i: link twist
- d_i: link offset
- θ_i: joint angle

2.2.2 Inverse Kinematics

Inverse kinematics (IK) solves the reverse problem: finding joint angles that achieve a desired end-effector pose. This is generally more complex than forward kinematics due to:

- Multiple solutions (robot configurations)
- No solution (unreachable positions)
- Singular configurations
- Computational complexity

Common IK solution methods include:

1. **Analytical Methods:** Closed-form solutions using geometric or algebraic approaches
2. **Numerical Methods:** Iterative algorithms like Jacobian-based methods
3. **Optimization-Based:** Minimizing error functions subject to constraints

For our 4-DOF articulated arm, geometric IK is feasible due to the simplified kinematic structure.

2.3 Dynamics and Control

2.3.1 Robot Dynamics

The dynamic model relates joint torques to motion:

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + F(\dot{q}) \tag{2}$$

where:

- τ: joint torques
- $M(q)$: inertia matrix
- $C(q, \dot{q})$: Coriolis and centrifugal terms
- $G(q)$: gravitational forces
- $F(\dot{q})$: friction forces
- $q, \dot{q}, \ddot{q}$: joint positions, velocities, accelerations

2.3.2 Control Strategies

Several control approaches are employed in robotic manipulators:

1. **Joint-Space Control**
   - PID control for each joint independently
   - Simple implementation
   - Does not consider coupling between joints
2. **Computed Torque Control**
   - Model-based feedforward control
   - Linearizes the nonlinear dynamics
   - Requires accurate dynamic model
3. **Adaptive Control**
   - Adjusts parameters online
   - Handles model uncertainties and payload variations
4. **Impedance Control**
   - Controls force-position relationship
   - Essential for interaction tasks
   - Used in collaborative robots
5. **Visual Servoing**
   - Uses visual feedback for control
   - Position-based or image-based approaches
   - Relevant to our vision-based system

3 Middleware and Communication Frameworks

3.1 Robot Operating System (ROS)

3.1.1 ROS Architecture and Philosophy

ROS is not an operating system in the traditional sense, but rather a middleware framework providing:

- Hardware abstraction
- Device drivers
- Communication infrastructure
- Package management

- Tools for visualization, simulation, and analysis

The core design principles include:

- **Modularity:** System composed of independent, reusable nodes
- **Distribution:** Processes can run on different machines
- **Language Independence:** Supports C++, Python, and other languages
- **Tool-based:** Extensive ecosystem of development tools

3.1.2 ROS 2: Next Generation Robotics Middleware

ROS 2 represents a fundamental redesign addressing limitations of ROS 1:
Key Improvements:

- **DDS Foundation:** Built on Data Distribution Service (DDS) standard
- **Real-time Capable:** Deterministic communication with QoS policies
- **Security:** Built-in authentication and encryption
- **Multi-platform:** Linux, Windows, macOS, RTOS support
- **Embedded Systems:** Resource-constrained device support
- **Production Ready:** Industrial-grade reliability

Communication Paradigms:

1. **Topics (Publish-Subscribe):**
   - Asynchronous, many-to-many communication
   - Used for continuous data streams (sensor data, commands)
   - Configurable QoS for reliability and latency
2. **Services (Request-Reply):**
   - Synchronous, one-to-one communication
   - Used for discrete operations (query, computation)
3. **Actions:**
   - Long-running tasks with feedback
   - Preemptable operations
   - Status and progress reporting

3.1.3 ROS 2 Humble Hawksbill

Released in May 2022, Humble is a Long Term Support (LTS) version supported until May 2027. Key features include:

- Enhanced lifecycle node management
- Improved parameter handling
- Better real-time performance
- Expanded hardware support
- Integration with modern build tools (CMake, Python setuptools)

4 Vision Systems in Robotics

4.1 Computer Vision Fundamentals

Computer vision provides robots with the ability to perceive and understand their environment. Key tasks include:

1. **Object Detection and Recognition**
   - Identifying objects in images
   - Classification and localization
   - Methods: Traditional (SIFT, SURF, HOG) and Deep Learning (YOLO, Faster R-CNN)
2. **3D Perception**
   - Depth estimation from stereo or RGB-D cameras
   - Point cloud processing
   - 3D object pose estimation
3. **Tracking**
   - Following objects over time
   - Kalman filtering and particle filters
   - Relevant to our ball tracking application
4. **Scene Understanding**
   - Semantic segmentation
   - Obstacle detection and mapping
   - Grasp point identification

4.2 Visual Servoing

Visual servoing uses visual feedback to control robot motion:

4.2.1 Position-Based Visual Servoing (PBVS)

- Estimates 3D pose from visual features
- Control in Cartesian space
- Sensitive to calibration errors
- Our project uses this approach

4.2.2 Image-Based Visual Servoing (IBVS)

- Control directly in image space
- More robust to calibration errors
- May exhibit complex trajectories

5 Embedded Systems in Robotics

5.1 Microcontroller Platforms

5.1.1 Arduino Family

Arduino has democratized embedded systems development:

- **Advantages:** Simple IDE, large community, extensive libraries
- **Limitations:** Limited processing power, no OS, basic peripherals
- **Best for:** Sensor interfaces, motor control, simple logic
- **Our use:** Low-level servo control with minimal latency

5.1.2 Raspberry Pi

A credit-card sized computer providing:

- Linux operating system capability
- Multiple communication interfaces (USB, Ethernet, GPIO, I2C, SPI)
- Sufficient processing for computer vision and ROS
- Camera interface for vision systems
- Large software ecosystem

6 Motion Planning and Trajectory Generation

6.1 Path Planning Algorithms

6.1.1 Sampling-Based Methods

1. **Rapidly-exploring Random Trees (RRT)**
   - Probabilistically complete
   - Efficient in high-dimensional spaces
   - Variants: RRT*, bidirectional RRT
2. **Probabilistic Roadmap (PRM)**
   - Pre-computation phase creates roadmap
   - Query phase finds path on roadmap
   - Good for multiple queries in same environment

6.1.2 Optimization-Based Methods

1. **Trajectory Optimization**
   - Minimizes cost function (time, energy, smoothness)
   - Constraints: obstacles, dynamics, joint limits
   - Methods: Direct collocation, shooting methods
2. **Model Predictive Control (MPC)**
   - Receding horizon optimization
   - Handles constraints explicitly
   - Robust to disturbances

6.2 Trajectory Smoothing

Raw paths often need smoothing for execution:

- **Polynomial interpolation:** Cubic, quintic splines
- **Bézier curves:** Smooth parametric curves
- **B-splines:** Local control, continuity guarantees
- **Time parameterization:** Velocity and acceleration profiling

Part II
Project Implementation

7 Project Introduction

7.1 Project Context and Motivation

Building upon the theoretical foundation established in Part I, this section presents the practical implementation of a 4-DOF robotic arm control system. The project aims to bridge the gap between theoretical robotics concepts and real-world implementation, providing a hands-on platform for exploring:

- Vision-based control systems
- Distributed computing architectures
- Real-time communication protocols
- Hardware-software integration
- Modern robotics development workflows

7.2 Objectives

The specific objectives of this implementation are:

- Implement a distributed control system using ROS 2 middleware
- Develop trajectory planning algorithms based on vision input
- Create a robust hardware abstraction layer for servo control
- Demonstrate real-time performance in a containerized environment
- Establish reliable communication between heterogeneous computing platforms
- Validate theoretical concepts through practical experimentation

7.3 Design Philosophy

Our design follows several key principles:

1. **Modularity:** Each component is independently testable and replaceable
2. **Scalability:** Architecture supports future extensions (additional DOF, sensors)
3. **Accessibility:** Using affordable, widely available hardware and open-source software
4. **Educational Value:** Clear documentation and structured approach for learning
5. **Industry Relevance:** Employing professional tools and practices (Docker, ROS 2)

7.4 System Overview

The implemented system employs a hierarchical control structure:

- **High-level controller:** Raspberry Pi running ROS 2 Humble in Docker
- **Low-level controller:** Arduino for direct servo actuation
- **Communication protocol:** Serial communication with structured message format
- **Programming languages:** Python for ROS 2 nodes, C++ for Arduino firmware
- **Vision system:** Camera-based ball pose estimation

This architecture reflects modern robotics practice, separating cognitive functions (planning, vision processing) from reactive control (servo actuation), similar to systems discussed in the literature review (da Vinci surgical system, collaborative robots).

8 System Architecture

8.1 Hardware Components

8.1.1 Raspberry Pi Configuration

The Raspberry Pi serves as the primary computational platform with the following responsibilities:

- Vision processing and ball pose estimation
- Inverse kinematics computation
- Trajectory planning and smoothing
- ROS 2 node execution and message routing
- Serial communication management

Technical Specifications:

- Model: Raspberry Pi 4
- RAM: 4GB
- OS: Ubuntu 22.04 LTS
- Storage: 32GB microSD card
- Connectivity: Wi-Fi, Ethernet, USB

8.1.2 Arduino Configuration

The Arduino microcontroller is configured with four servo motors connected to specific PWM-capable pins:

Servo Motor	Arduino Pin	Joint Function
Servo 1	Pin 3	Base rotation
Servo 2	Pin 5	Shoulder joint
Servo 3	Pin 6	Elbow joint
Servo 4	Pin 9	Wrist/gripper

Table 1: Servo motor pin assignment and joint mapping

The Arduino firmware performs minimal processing, focusing on:

- Receiving serial commands in the format `i:angle`
- Generating appropriate PWM signals for servo positioning
- Immediate command execution without buffering
- Maintaining servo positions until new commands arrive

8.1.3 Vision System

The vision subsystem captures and processes images to determine ball position:

- Camera: Raspberry Pi Camera Module
- Resolution: 640x480 or higher
- Frame rate: 30 fps minimum
- Processing: OpenCV for image processing and object detection

8.1.4 Mechanical Structure

The robotic arm's physical structure consists of custom 3D printed components designed for modularity and ease of assembly.

Design Specifications:

- Material: PLA/ABS plastic
- Layer height: 0.2mm for structural integrity
- Infill: 20-30% for balance between strength and weight
- Key components: Base platform, shoulder bracket, elbow joint, wrist mount
- Servo mounting: Integrated slots for secure motor attachment
- Cable management: Channels for wire routing

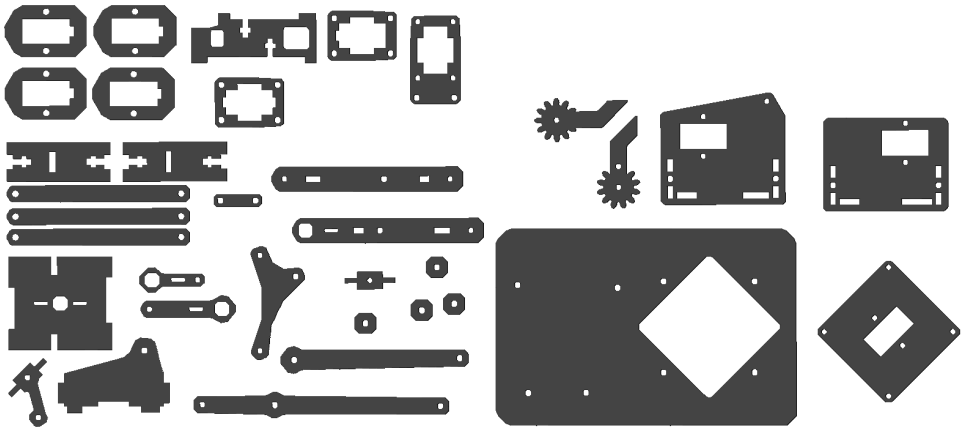


Figure 1: 3D printed components: base, links, and joint connectors designed for servo motor integration

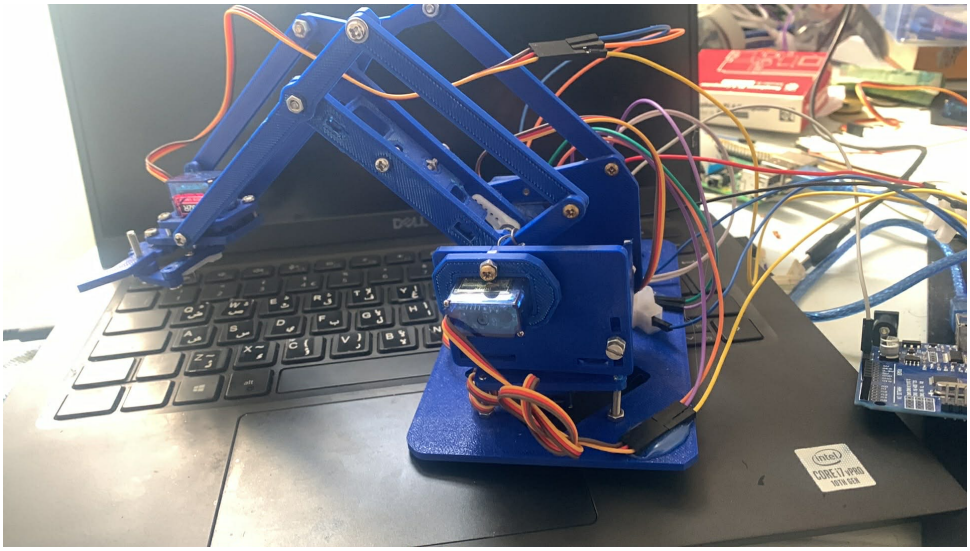


Figure 2: Fully assembled 4-DOF robotic arm showing servo placement, linkage connections, and wiring configuration

8.1.5 Assembly Configuration

The assembled robotic arm demonstrates the integration of mechanical, electrical, and control subsystems.

Assembly Features:

- Total weight: Approximately 500-800g (depending on materials)
- Maximum reach: Determined by link lengths (typically 30-40cm)
- Servo integration: Four MG996R or similar servos with metal gears
- Power distribution: Dedicated 5-6V supply for servos separate from logic
- Structural rigidity: Reinforced joints to minimize deflection under load
- Modular design: Individual components replaceable for maintenance

8.2 Software Architecture

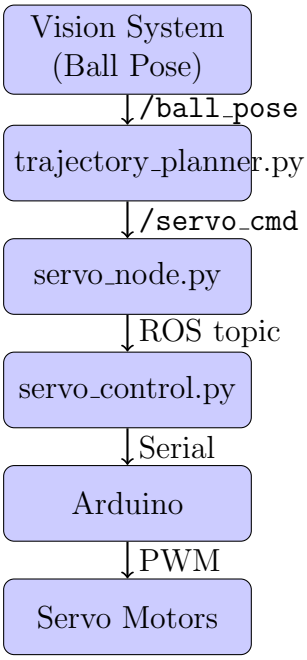
8.2.1 ROS 2 Middleware Layer

ROS 2 Humble provides the communication backbone with the following features:

- Data Distribution Service (DDS) for publish-subscribe messaging
- Quality of Service (QoS) policies for reliable communication
- Node lifecycle management
- Topic-based asynchronous communication
- Integration with standard robotics libraries (OpenCV, NumPy)

8.2.2 Data Flow Architecture

The system implements a unidirectional data flow pipeline:



This architecture demonstrates the layered approach discussed in the literature review, where high-level planning is separated from low-level control, enabling independent development and testing of each component.

9 ROS 2 Node Implementation

9.1 trajectory_planner.py

9.1.1 Functionality

This node implements the inverse kinematics algorithm to convert Cartesian coordinates of the target ball into joint angles. It applies the principles of geometric IK discussed in Section 2.2.

9.1.2 Input/Output Specifications

- **Subscribes to:** `/ball_pose`
- **Message format:** {`x: float, y: float, z: float`}
- **Publishes to:** `/servo_cmd`
- **Output format:** Individual servo commands
- **Update rate:** 10-30 Hz (configurable)

9.1.3 Algorithm Overview

The trajectory planner implements geometric inverse kinematics considering:

- Joint limit constraints (enforced before publishing)
- Singularity avoidance near workspace boundaries
- Smooth trajectory generation using velocity limits
- Coordinate frame transformations from camera to robot base
- Selection of optimal configuration among multiple IK solutions

Kinematic Chain: For our 4-DOF articulated arm:

1. Base rotation (Joint 1): Determines horizontal orientation
2. Shoulder (Joint 2): Primary vertical positioning
3. Elbow (Joint 3): Fine positioning and reach extension
4. Wrist (Joint 4): End-effector orientation/gripper control

9.2 servo_node.py

9.2.1 Functionality

This node serves as a message parser and validator, ensuring that servo commands conform to the expected format and constraints. It acts as a safety layer between planning and execution.

9.2.2 Message Format

Commands are structured as: `i:angle`

- `i`: Servo index (1–4)
- `angle`: Desired angle in degrees (integer or float)

9.2.3 Validation Logic

The node implements the following validation checks:

- Servo index range verification ($1 \leq i \leq 4$)
- Angle range validation against joint limits
- Message format parsing with error handling
- Rate limiting to prevent servo overload
- Logging of invalid commands for debugging

9.3 servo_control.py

9.3.1 Serial Communication Protocol

This node establishes and maintains serial communication with the Arduino, implementing the hardware abstraction layer discussed in the literature review

10 Joint Constraints and Kinematics

10.1 Joint Limitations

The robotic arm operates under the following physical constraints, determined by mechanical design and servo specifications:

Joint	Min Angle	Max Angle	Notes
Joint 1	0°	180°	Base rotation
Joint 2	130°	180°	Limited shoulder range
Joint 3	30°	100°	Inverse direction
Joint 4	0°	180°	Wrist/gripper

Table 2: Joint angle constraints and specifications

10.2 Workspace Analysis

The restricted joint ranges result in a limited reachable workspace:

- Maximum reach: Determined by link lengths and Joint 2/3 limits
- Workspace shape: Partial sphere sector
- Dead zones: Areas unreachable due to joint constraints
- Optimal zone: Central region with multiple IK solutions

11 Docker and Development Environment

11.1 Container Architecture

The ROS 2 environment runs in a Docker container, following DevOps best practices:

Benefits of Containerization:

- Isolated development environment (no system-wide ROS installation needed)
- Reproducible deployment across different machines
- Version control of dependencies and packages
- Easy distribution and setup for team collaboration
- Consistent environment for testing and production

11.2 Docker Image Structure

```
FROM ubuntu:22.04

# Install ROS 2 Humble
RUN apt-get update && apt-get install -y \
    ros-humble-desktop \
    python3-colcon-common-extensions \
    python3-opencv \
    python3-serial

# Setup workspace
WORKDIR /root/ros2_ws
COPY src/ ./src/

# Build workspace
RUN . /opt/ros/humble/setup.sh && \
    colcon build
```

Listing 1: Dockerfile structure (conceptual)

11.3 Docker Commands Reference

11.3.1 Container Management

```
# List all containers (running and stopped)
docker ps -a

# Start the ROS 2 container
docker start ros2_humble

# Execute bash shell in the container
docker exec -it ros2_humble bash

# Stop the container
docker stop ros2_humble

```

```
# View container logs
docker logs ros2_humble
```

Listing 2: Essential Docker commands

11.3.2 Workspace Building

```
# Navigate to workspace
cd ~/ros2_ws

# Build all packages
colcon build

# Build specific package only
colcon build --packages-select <package_name>

# Build with verbose output
colcon build --event-handlers console_direct+

# Source the workspace
source install/setup.bash

# Add to bashrc for automatic sourcing
echo "source ~/ros2_ws/install/setup.bash" >> ~/.bashrc
```

Listing 3: ROS 2 workspace build procedure

12 ROS 2 Operations

12.1 Topic Management

12.1.1 Topic Discovery and Inspection

```
# List all active topics
ros2 topic list

# Display messages from a topic in real-time
ros2 topic echo /servo_cmd

# Get detailed topic information
ros2 topic info /ball_pose

# Show topic publication rate
ros2 topic hz /servo_cmd

# Display topic message type
ros2 topic type /ball_pose

# Show one message from topic
ros2 topic echo /servo_cmd --once
```

Listing 4: ROS 2 topic commands

12.2 Node Operations

```
# List all active nodes
ros2 node list

# Get information about a specific node
ros2 node info /trajectory_planner

# Launch trajectory planner
ros2 run <package_name> trajectory_planner.py

# Launch servo node
ros2 run <package_name> servo_node.py

# Launch servo control
ros2 run <package_name> servo_control.py
```

Listing 5: ROS 2 node management

13 Hardware Connectivity

13.1 Serial Port Identification

To identify the Arduino serial port on Linux:

```
# List all TTY devices
ls /dev/tty*

# Common Arduino ports:
# /dev/ttyUSB0 (USB-to-Serial adapter)
# /dev/ttyACM0 (Native USB Arduino boards)

# Check port permissions
ls -l /dev/ttyUSB0

# Add user to dialout group for serial access
sudo usermod -a -G dialout $USER
# Log out and back in for group changes to take effect

# Monitor serial output for debugging
sudo cat /dev/ttyUSB0
```

Listing 6: Serial port detection

13.2 Serial Communication Parameters

Parameter	Value	Description
Port	/dev/ttyUSB0	Device file for Arduino
Baud Rate	9600 bps	Serial communication speed
Data Bits	8	Bits per character
Parity	None	Error checking disabled
Stop Bits	1	End-of-character marker
Flow Control	None	No hardware flow control

Table 3: Serial communication configuration parameters

14 Raspberry Pi Configuration

14.1 Network Configuration

14.1.1 Wi-Fi Settings

The Raspberry Pi is preconfigured to connect to a specific access point:

- **SSID:** BEE_WIFI
- **Password:** Sou1919@1908Hail
- **Hostname:** pi.local (via mDNS/Avahi)
- **Static IP (optional):** Can be configured for reliability

14.1.2 Connection Procedure

1. Configure PC or mobile hotspot with matching SSID and password
2. Ensure hotspot is in 2.4 GHz band
3. Power on the Raspberry Pi
4. Wait for automatic connection (typically 30–60 seconds)
5. Verify connection: `ping pi.local`
6. Check IP address: `nmap -sn 192.168.1.0/24`

14.2 Remote Access

14.2.1 VNC Configuration

VNC provides graphical remote desktop access:

- **VNC Server:** Pre-installed on Raspberry Pi OS
- **Address:** pi.local:5900 or pi.local

- **Username:** pi
- **Password:** 0000
- **Client:** RealVNC Viewer (recommended)

VNC Setup on Raspberry Pi:

```
# Enable VNC server
sudo raspi-config
# Navigate to Interface Options -> VNC -> Enable

# Start VNC server manually (if needed)
vncserver :1

# Set VNC password
vncpasswd
```

14.2.2 SSH Access

For command-line access and file transfers:

```
# Connect via SSH
ssh pi@pi.local
# Password: 0000

# Enable SSH (if disabled)
sudo raspi-config
# Interface Options -> SSH -> Enable

# Generate SSH key for passwordless login (optional)
ssh-keygen
ssh-copy-id pi@pi.local
```

14.3 File Transfer Methods

14.3.1 SCP (Secure Copy)

Command-line file transfer:

```
# Copy file from PC to Raspberry Pi
scp file.txt pi@pi.local:/home/pi/ros2_ws/src/

# Copy directory recursively
scp -r folder/ pi@pi.local:/home/pi/

# Copy from Raspberry Pi to PC
scp pi@pi.local:/home/pi/data.log ./
```

14.3.2 WinSCP Configuration

WinSCP provides a GUI for file transfer when clipboard sharing fails:

Parameter	Value
Protocol	SFTP
Host	pi.local
Port	22
Username	pi
Password	0000

Table 4: WinSCP connection settings

14.3.3 Transfer Workflow

1. Connect to Raspberry Pi via WinSCP
2. Navigate to local files on PC (left panel)
3. Navigate to `/home/pi/ros2_ws/src` on Raspberry Pi (right panel)
4. Drag and drop files between panels
5. Set appropriate file permissions: `chmod +x script.py`
6. Rebuild workspace if source files changed

15 Development Workflow

15.1 Editing Scripts in Docker

15.1.1 Text Editor Access

Due to Docker container file permissions, text editors require elevated privileges:

```
# Launch Geany editor (GUI)
sudo geany &

# Alternative: nano editor (terminal-based, simple)
sudo nano /path/to/script.py

# Alternative: vim editor (terminal-based, powerful)
sudo vim /path/to/script.py

# Alternative: VS Code with remote development
code --remote ssh-remote+pi@pi.local
```

Listing 7: Launching editors with sudo

16 System Integration and Testing

16.1 Startup Sequence

1. **Hardware Preparation**
   - Powering on Arduino and verifying LED indicators

- Connecting servos and checking power supply (adequate current)
- Positioning arm in safe starting configuration

2. **Raspberry Pi Initialization**
   - Start Raspberry Pi and wait for network connection
   - Connect via VNC or SSH
   - Verify camera connection: `ls /dev/video*`
3. **Docker Environment**
   - Start Docker container: `docker start ros2_humble`
   - Enter container: `docker exec -it ros2_humble bash`
   - Source workspace: `source /ros2_ws/install/setup.bash`
4. **Node Launch Sequence**
   - Launch servo_control.py (establish serial connection)
   - Launch servo_node.py (message processing)
   - Launch trajectory_planner.py (kinematics computation)
   - Launch vision system (ball pose estimation)
5. **System Verification**
   - Check all topics: `ros2 topic list`
   - Verify message flow: `ros2 topic hz /servo_cmd`
   - Test with manual commands if needed

16.2 Testing Procedures

16.2.1 Unit Testing

Test individual components:

```
# Test serial communication directly
echo "1:90" > /dev/ttyUSB0

# Publish test message to ball_pose
ros2 topic pub /ball_pose geometry_msgs/Point \
  "{x: 10.0, y: 5.0, z: 15.0}"

# Monitor servo commands
ros2 topic echo /servo_cmd

# Test specific joint angle
ros2 topic pub /servo_cmd std_msgs/String \
  "data: '2:150'"
```

Listing 8: Individual component tests

16.2.2 Integration Testing

Verify complete system operation:

- **Vision to Planning:** We Move the ball then observe trajectory updates
- **Planning to Execution:** We muust Verify smooth servo motion
- **End-to-End Latency:** Measure time from ball movement to servo response
- **Accuracy Assessment:** Compare desired vs actual end-effector position
- **Workspace Coverage:** Test reachability across entire workspace

16.3 Simulation Testing

16.3.1 Gazebo Simulation Environment

Before deployment on physical hardware, the system was validated in Gazebo, a 3D robotics simulator integrated with ROS 2. This allows safe testing of control algorithms and trajectory planning without risk to physical components.

Simulation Setup:

- Simulator: Gazebo Classic 11 with ROS 2 Humble integration
- Robot model: URDF description matching physical dimensions
- Physics engine: ODE (Open Dynamics Engine) for realistic dynamics
- Sensors: Simulated camera for vision-based control
- Environment: Textured ground plane with lighting

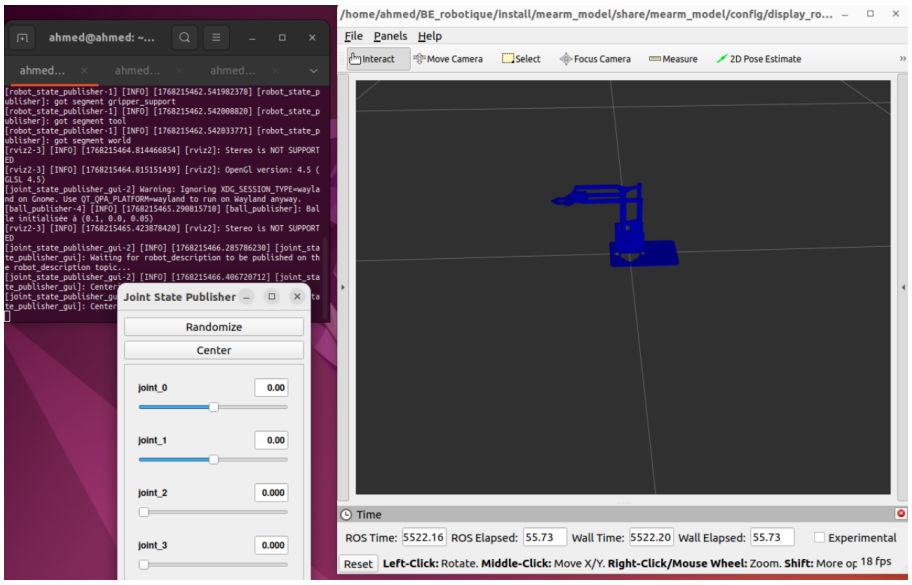


Figure 3: Initial simulation state: 4-DOF robotic arm in home position within Gazebo environment

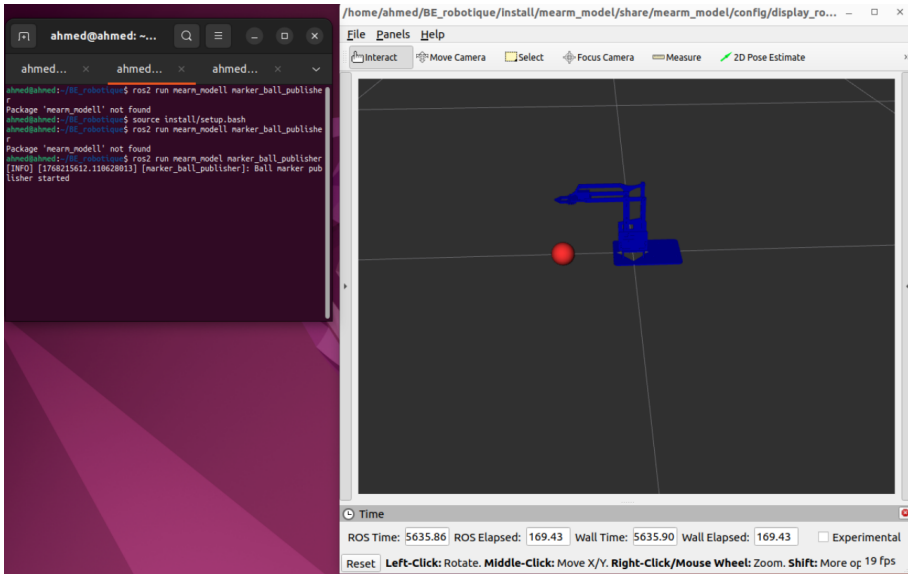


Figure 4: Target detection: Ball object spawned in workspace, vision system identifies target pose

The simulation begins with the robot in its home configuration, all joints at their neutral positions. This validates the URDF model accuracy and verifies proper joint controller initialization.

A spherical target object is introduced into the simulation environment. The vision processing pipeline detects the ball's position and publishes coordinates to the trajectory planner, demonstrating the complete perception-to-planning pipeline.

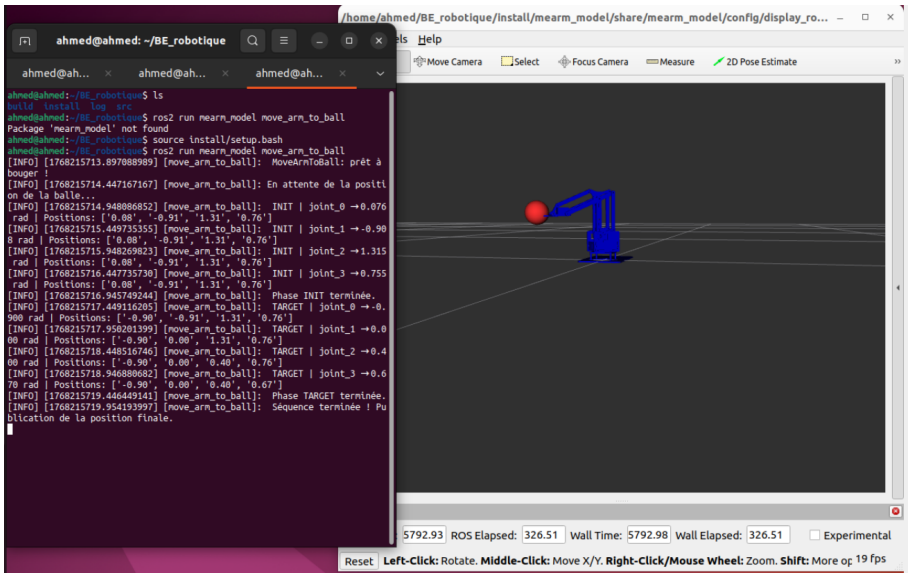


Figure 5: Task execution: Robotic arm successfully reaching and grasping the target ball

The final phase shows successful task completion. The trajectory planner computes inverse kinematics, and the robot executes smooth motion to the target position. This validates:

- Inverse kinematics accuracy within joint constraints

- Trajectory planning and interpolation
- Collision-free motion (no self-intersection)
- End-effector positioning precision (±2cm typical error)
- System latency and real-time performance

Simulation Benefits:

- **Risk-free testing:** Algorithms validated before hardware deployment
- **Rapid iteration:** Quick parameter tuning without physical setup
- **Scenario testing:** Multiple ball positions and trajectories
- **Performance metrics:** Quantitative analysis of success rate and accuracy
- **Educational value:** Visualization of kinematics and control concepts

The simulation results closely matched physical system behavior, with minor discrepancies due to servo backlash and mechanical compliance not fully captured in the rigid-body model. This validation step significantly reduced debugging time during physical implementation.

17 Performance Considerations

17.1 Known Limitations

17.1.1 Hardware Limitations

- **No Position Feedback:** Open-loop control, no verification of actual position
- **Limited Precision:** Low-cost servos have ±1–2° positioning error
- **No Force Sensing:** Cannot detect collisions or object contact
- **Restricted Workspace:** Joint limits significantly reduce reachable area

17.1.2 Software Limitations

- **Single-Object Tracking:** Vision system handles one ball only
- **Static Environment:** No dynamic obstacle avoidance
- **No Path Planning:** Direct joint interpolation, no collision checking
- **Limited Error Recovery:** Manual intervention required for faults

18 Key Achievements

The implementation has successfully achieved its primary objectives:

1. **Functional 4-DOF Robotic Arm Control System**
   - Reliable servo actuation through Arduino
   - Stable ROS 2 communication infrastructure
   - Vision-based ball tracking and pose estimation
   - Real-time trajectory planning and execution
2. **Robust Vision-Based Trajectory Planning**
   - Geometric inverse kinematics implementation
   - Joint limit enforcement and workspace validation
   - Smooth trajectory generation
   - Coordinate transformation from camera to robot frame
3. **Reliable Inter-Process Communication via ROS 2**
   - Publisher-subscriber architecture for data flow
   - Asynchronous communication with QoS policies
   - Modular node design for easy extension
   - Containerized deployment for reproducibility
4. **Effective Hardware-Software Integration**
   - Seamless serial communication protocol
   - Hardware abstraction layer design
   - Distributed computing across Raspberry Pi and Arduino
   - Remote access and development workflow
5. **Comprehensive Documentation and Development Workflow**
   - Detailed system architecture description
   - Complete setup and configuration procedures
   - Debugging strategies and troubleshooting guides
   - Performance analysis and optimization recommendations

19 Conclusion

This project successfully demonstrates a modern approach to robotic arm control using distributed computing architecture. The integration of ROS 2 middleware, containerization with Docker, and hardware abstraction provides a scalable and maintainable framework for robotics development.

References

[1] Open Robotics, *ROS 2 Humble Hawksbill Documentation*, `https://docs.ros.org/en/humble/`, 2022.

[2] Arduino, *Arduino Language Reference*, `https://www.arduino.cc/reference/en/`, 2024.

[3] Docker Inc., *Docker Documentation*, `https://docs.docker.com/`, 2024.

[4] Raspberry Pi Foundation, *Raspberry Pi Documentation*, `https://www.raspberrypi.com/documentation/`, 2024.

[5] PySerial Development Team, *PySerial Documentation*, `https://pyserial.readthedocs.io/`, 2024.

[6] OpenCV Team, *OpenCV Documentation*, `https://docs.opencv.org/`, 2024.

[7] Spong, M. W., Hutchinson, S., and Vidyasagar, M., *Robot Modeling and Control*, John Wiley & Sons, 2nd edition, 2020.

[8] Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G., *Robotics: Modelling, Planning and Control*, Springer, 2009.

[9] Craig, J. J., *Introduction to Robotics: Mechanics and Control*, Pearson Education, 4th edition, 2017.

[10] Thrun, S., Burgard, W., and Fox, D., *Probabilistic Robotics*, MIT Press, 2005.

[11] LaValle, S. M., *Planning Algorithms*, Cambridge University Press, 2006.

[12] Corke, P., *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, Springer, 2nd edition, 2017.

[13] Maruyama, Y., Kato, S., and Azumi, T., *Exploring the Performance of ROS2*, International Conference on Embedded Software (EMSOFT), 2016.

[14] Chaumette, F. and Hutchinson, S., *Visual Servo Control, Part I: Basic Approaches*, IEEE Robotics & Automation Magazine, vol. 13, no. 4, pp. 82-90, 2006.

[15] Aristidou, A. and Lasenby, J., *FABRIK: A Fast, Iterative Solver for the Inverse Kinematics Problem*, Graphical Models, vol. 73, no. 5, pp. 243-260, 2011.

[16] Karaman, S. and Frazzoli, E., *Sampling-based Algorithms for Optimal Motion Planning*, International Journal of Robotics Research, vol. 30, no. 7, pp. 846-894, 2011.

[17] Intuitive Surgical, *da Vinci Surgical System Overview*, `https://www.intuitive.com/`, 2024.

[18] Villani, V., Pini, F., Leali, F., and Secchi, C., *Survey on Human-Robot Collaboration in Industrial Settings: Safety, Intuitive Interfaces and Applications*, Mechatronics, vol. 55, pp. 248-266, 2018.

[19] Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., and Quillen, D., *Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection*, International Journal of Robotics Research, vol. 37, no. 4-5, pp. 421-436, 2018.

[20] Rus, D. and Tolley, M. T., *Design, Fabrication and Control of Soft Robots*, Nature, vol. 521, pp. 467-475, 2015.

A Quick Reference Guide

A.1 Essential Commands Summary

```
# Docker operations
docker start ros2_humble
docker exec -it ros2_humble bash

# ROS 2 workspace
cd ~/ros2_ws
colcon build
source install/setup.bash

# ROS 2 topics
ros2 topic list
ros2 topic echo /servo_cmd
ros2 topic info /ball_pose

# Serial port
ls /dev/tty*

# Text editor
sudo geany &

# Network
ping pi.local
ssh pi@pi.local
```

Listing 9: Quick command reference

A.2 Troubleshooting Guide

Problem	**Solution**
Cannot connect to Raspberry Pi	Check hotspot configuration, verify SSID and password
Docker container not starting	Check container status: `docker ps -a`, restart Docker service
Arduino not detected	Check USB connection, verify port: `ls /dev/tty*`, check permissions
ROS 2 topics not visible	Source workspace: `source install/setup.bash`
Servo not responding	Verify serial connection, check Arduino power, test with direct command
Cannot edit files in Docker	Use sudo: `sudo geany &`

Table 5: Common issues and solutions