REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

الجمهورية الجزائرية الديمقراطية الشعبية

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

HIGHER SCHOOL IN APPLIED SCIENCES
--T L E M C E N--

وزارة التعليم العالي والبحث العلمي

المدرسة العليا في العلوم التطبيقية
-تلمسان-

Mémoire de fin d'étude

Pour l'obtention du diplôme de Master

Filière : Automatique
Spécialité : Automatique

Présenté par : MEKELLECHE Nadjib
BERREHOU Abdelwahab Djacim

Thème

Commande d'un robot manipulateur mobile

Soutenu publiquement, le 26 / 09 / 2024, devant le jury composé de :

M. F.ARICHI	MCA	ESSA. Tlemcen	Président
M. B.CHERKI	Professeur	ESSA.Tlemcen	Directeur de mémoire
M. S.M.ABDI	MCB	ESSA.Tlemcen	Examinateur 1
M. H.MEGNAFI	MCA	ESSA.Tlemcen	Examinateur 2
M. B.LASSOUANI	Ingénieur	ALGERIE TELECOM	Partenaire socioéconomique
M. R.A.ADJIM	Ingénieur de laboratoire	ESSA.Tlemcen	Invité 1

Année universitaire : 2023 / 2024

Dedications

I dedicate this modest work to:

My dear parents, whose unwavering love and countless sacrifices have shaped my journey. Your steadfast support and heartfelt prayers have been my guiding light throughout my studies, and for that, I am eternally grateful.

To my younger sister, for her unwavering support and encouragement. May you chase your dreams and achieve great things.

In loving memory of my paternal grandparents, whose wisdom and strength continue to inspire me, and to my maternal grandparents, whose love and guidance have been a foundation in my life.

To my entire family, thank you for your constant encouragement, your sage advice, and the strength you provide. Your belief in me has fueled my ambitions and helped me overcome every challenge.

To all my friends, your support has been invaluable. The bonds we share and the sincere friendships we cultivate enrich my life and motivate me to strive for excellence.

To everyone dear to me, thank you for being part of my journey. Your love and support mean the world to me.

And finally, to you, the reader of this dedication: your presence in my life has been a source of strength. Thank you for standing by me during times of need. Your kindness and encouragement are deeply appreciated.

MEKELLECHE Nadjib

To my mother, who has been with me every step of this journey since the very first day, for the past 17 years. Your love and constant presence have been my source of strength and guidance.

To my father, who has sacrificed so much for me, always ensuring I had what I needed to succeed. Your dedication has not gone unnoticed, and I am forever grateful.

To my brothers, who have been my strength when I needed it most, always standing by me.

To my sisters, whose support and cheers have lifted me up, pushing me to keep going even when things were tough. To my friends, Adel, Zaki, and Wail, my brothers from another mothers, your friendship has meant the world to me.

To that one person reading this dedication, thank you for being there for me when I needed.

BERREHOU Abdelwahab Djacim

Acknowledgments

We express our deep gratitude to Almighty God for guiding us in the accomplishment of this modest work.

We would like to express our gratitude to our families, and especially our parents, for their unconditional support and encouragement throughout our studies.

We sincerely thank **Professor CHERKI Brahim** for his valuable supervision, his enlightened advice and his constant support throughout this project. His guidance was essential to overcome the technical challenges encountered.

We are deeply honored by the participation of **Dr. ARICHI Fayssal**, who chaired our thesis and honored our jury with his presence.

We would like to express our sincere thanks to **Dr. ABDI Sidi Mohammed** and **Dr. MEGNAFI Hicham** for agreeing to review this thesis and for their relevant and constructive comments.

We express our deep gratitude to **Mr. ADJIM Ramz-eddine Abderrezak**, engineer of FABLAB at the Higher School of Applied Sciences of Tlemcen, and **Mr. AISSANI Omar** for their valuable contribution in the realization of our work.

We thank **Mr. Brahim LASSOUANI** for taking the time to evaluate our project and for providing valuable feedback and guidance.

We would also like to warmly thank all our friends as well as all the people who contributed directly or indirectly to the realization of this work. Their support and collaboration were greatly appreciated.

ملخص

يستكشف هذا المشروع تصميم والتحكم وتنفيذ روبوت موجه متحرك، مع دمج المفاهيم النظرية والتطبيقات العملية. يتناول الأنظمة والمكونات الأساسية المعنية، ونمذجة حركات الروبوت وتفاعلاته، وتوليد وتخطيط مساراته. كما يناقش التقرير أيضًا محاكاة هذه الروبوتات للتحقق من أدائها ويبحث في الجوانب العملية للتنفيذ، بما في ذلك التكامل بين الأجهزة والبرمجيات.

Abstract

This project explores the design, control, and implementation of a mobile manipulator robot, integrating theoretical concepts and practical applications. It covers the essential systems and components involved, the modeling of the robot's movements and interactions, and the generation and planning of its trajectories. The report also addresses the simulation of these robots to validate their performance and discusses practical implementation aspects, including hardware and software integration.

Résumé

Ce projet explore la conception, le contrôle et la mise en uvre d'un robot manipulateur mobile, intégrant des concepts théoriques et des applications pratiques. Il couvre les systèmes et composants essentiels impliqués, la modélisation des mouvements et interactions du robot, ainsi que la génération et la planification de ses trajectoires. Le rapport aborde également la simulation de ces robots pour valider leur performance et discute des aspects pratiques de la mise en uvre, y compris l'intégration matérielle et logicielle.

Contents

List of Figures

List of Tables

Abbreviations

AGV	Automated Guided Vehicle
AMR	Autonomous Mobile Robot
EEF	End Effector
DC	Direct Current
LIDAR	Light Detection and Ranging
ROS	Robot Operating System
RViz	ROS Visualization
URDF	Unified Robot Description Format
GUI	Graphical User Interface
IK	Inverse Kinematics
FK	Forward Kinematics
DOF	Degrees of Freedom
PID	Proportional-Integral-Derivative
PWM	Pulse Width Modulation
SLAM	Simultaneous Localization and Mapping
MPC	Model Predictive Control
RRT	Rapidly-Exploring Random Tree
A*	A star search algorithm
AI	Artificial Intelligence
QR	Quick Response

General Introduction

The control of mobile manipulator robots integrates complex systems involving various disciplines of robotics, control theory, and software engineering. This project report presents a comprehensive exploration of the essential aspects of mobile manipulator robots, organized into several core domains.
The discussion begins with an exploration of core concepts in robotic control, addressing essential control strategies. The report progresses to command and control strategies tailored to mobile robots, manipulator arms, and mobile manipulators.
Next, the focus shifts to modeling, addressing the kinematic modeling of robotic systems. Special attention is given to mobile robots, particularly those utilizing Mecanum wheels, with a detailed analysis of kinematic constraints and motion approximation techniques. For robotic arms, the discussion includes Denavit-Hartenberg parameters and kinematic modeling, encompassing forward and inverse kinematics, essential for accurate robotic motion planning.
The report then examines trajectory generation and planning, specifically polynomial-based trajectory generation, such as cubic and quintic polynomials, which are crucial for smooth and precise robot movements. It also includes an in-depth study of pathfinding algorithms, such as Dijkstra's and A*.
Simulation provides insights into the simulation of robotic systems using advanced tools like MATLAB and GAZEBO. This includes methodologies for simulating both manipulator arms and mobile robots, essential for validating theoretical models and control strategies in virtual environments.
Implementation aspects cover practical considerations for implementing robotic control systems. This includes the use of platforms such as Raspberry Pi and Arduino, communication protocols, and the integration of OpenCV for computer vision tasks. The discussion also explores various graphical user interfaces designed for controlling robot manipulators and mobile systems, pathfinding visualization, QR code detection, and real-time system integration.

Chapter 1

Robotics : Systems and Components

Robotics, a field at the intersection of engineering and computer science, has evolved significantly over the decades. The historical development of robotics showcases a journey from rudimentary mechanical devices to sophisticated autonomous systems. Early robots were primarily mechanical constructs with limited functionality, often used in repetitive industrial tasks. Over time, advancements in electronics, computing, and artificial intelligence have transformed robots into versatile machines capable of complex behaviors and interactions.

The definition and classification of robots encompass a broad spectrum of types, including mobile robots, manipulator arms, and mobile manipulator robots. Mobile robots are designed to move and navigate through various environments, leveraging sensors and control algorithms to perform tasks autonomously. Manipulator arms, on the other hand, are robotic systems with articulated joints and end-effectors used for tasks requiring precise manipulation and control. Mobile manipulator robots combine the attributes of mobile robots and manipulator arms, offering a unique capability to navigate and manipulate objects within their environment.

Command and control systems play a pivotal role in the functionality of robots. These systems are essential for interpreting sensor data, executing control algorithms, and coordinating robotic actions. The importance of command and control is evident in diverse applications such as industrial automation, where robots perform repetitive tasks with high precision; service robots that assist in domestic and healthcare settings; autonomous vehicles navigating complex traffic scenarios; and medical robotics enhancing surgical precision.

This chapter aims to introduce fundamental concepts in robotics command and control, exploring the integration of hardware, software, and theoretical frameworks. It provides a comprehensive overview of the principles underlying robotic systems, control architectures, sensor integration, and advanced strategies for mobile robots, manipulator arms, and mobile manipulators.

1.1 Theoretical Foundations of Robotic Control

Control theory forms the backbone of robotic systems, providing the principles and techniques necessary for managing and directing robot behaviors. Fundamental control principles include feedback, feedforward, and hybrid controls. Feedback control involves continuously adjusting the system based on the difference between desired and actual outputs, ensuring stability and accuracy. Feedforward control anticipates system changes and adjusts commands proactively to improve performance. Hybrid controls combine both feedback and feedforward mechanisms to leverage the strengths of each approach. Key concepts in control theory relevant to robotics are stability, controllability, and observability. Stability ensures that the robot's behavior remains predictable and within acceptable bounds. Controllability refers to the ability to direct the robot's state through input controls, while observability involves the capacity to infer the internal state of the robot from external measurements. These principles are crucial for designing robust and reliable robotic systems.

1.2 Command Architectures in Robotics

1.2.1 Centralized and Decentralized Control

Robotic systems can be designed with centralized or decentralized control architectures, each with distinct advantages and applications. Centralized control involves a single, central unit making decisions and issuing commands to all robot components. This approach simplifies coordination and management but may become a bottleneck in complex systems [1]. In contrast, decentralized control distributes decision-making across multiple units, enhancing robustness and scalability. Centralized control is often used in industrial robots where precise, synchronized movements are required, while decentralized control is common in swarm robotics, where individual robots operate autonomously and coordinate through local interactions.

1.2.2 Real-time Control Systems

Real-time control systems are critical for ensuring that robots respond promptly to environmental changes and achieve desired behaviors. Essential requirements for real-time operations include deterministic timing, low latency, and high reliability [2]. Techniques for ensuring real-time performance involve using dedicated real-time operating systems, optimizing control algorithms, and implementing efficient communication protocols. These systems are crucial for applications requiring immediate responses, such as autonomous driving and robotic surgery.

1.2.3 Adaptive and Predictive Control

Adaptive control strategies enable robots to handle uncertainties and variations in their operating environment. These strategies adjust control parameters in real-time based on observed changes, ensuring optimal performance under varying conditions. Model Predictive Control (MPC) is a sophisticated technique that uses a model of the robot to predict future states and optimize control inputs accordingly [3]. MPC is particularly useful for managing complex constraints and achieving precise control in dynamic environments.

1.3 Sensor Integration and Data Fusion

1.3.1 Types of Sensors in Robotics

Sensors are vital for robots to perceive and interact with their surroundings [4]. Common sensors include vision sensors (cameras), proximity sensors (ultrasonic, infrared), tactile sensors (force/torque sensors), and others. Each sensor type provides specific information crucial for robot functionality. Sensor accuracy and calibration are essential for reliable operation, as inaccuracies can lead to errors in perception and control.

1.3.2 Sensor Fusion Techniques

Sensor fusion involves combining data from multiple sensors to create a comprehensive understanding of the robot's environment. Techniques such as Kalman filtering [5] and particle filtering [6] are used to integrate sensor data, improving accuracy and robustness. Applications of sensor fusion include localization and navigation, where combined sensor inputs help estimate the robot's position and trajectory more precisely.

1.3.3 State Estimation

State estimation is a key component of robotic control, enabling robots to infer their internal states based on sensor measurements [7]. Filters such as the Kalman filter are commonly used to estimate states and predict future positions. Practical examples include mobile robots using state estimation for precise navigation and manipulator arms using it for accurate positioning.

1.4 Command and Control Strategies for Mobile Robots

1.4.1 Path Planning and Navigation

Path planning involves determining an optimal path for a mobile robot to follow from a starting point to a goal. Algorithms such as A*, Dijkstra, and Rapidly-exploring Random Trees (RRT) are commonly used for this purpose. Navigation strategies include localization, which involves determining the robot's position, mapping, which creates a representation of the environment, and Simultaneous Localization and Mapping (SLAM), which integrates both localization and mapping.

1.4.2 Motion Control

Motion control strategies manage the movement of mobile robots, ensuring they follow planned paths and respond to environmental changes. Techniques such as PID control, Model Predictive Control (MPC), and fuzzy logic control are used to regulate robot motion. Challenges in motion control include dealing with nonholonomic constraints and avoiding obstacles, which require advanced algorithms and sensor integration.

1.4.3 Communication and Coordination in Multi-Robot Systems

Multi-robot systems involve multiple robots working together to achieve a common goal. Communication protocols and strategies are essential for coordinating actions and sharing information among robots. Distributed control and decentralized decision-making enable robots to collaborate effectively, with each robot making decisions based on local information and interactions.

1.5 Command and Control Strategies for Manipulator Arms

1.5.1 Kinematic Control

Kinematic control focuses on the precise movement of manipulator arms. Forward kinematics involves calculating the end-effector's position based on joint parameters, while inverse kinematics determines the necessary joint angles to achieve a desired end-effector position.

1.5.2 Dynamic Control

Dynamic control addresses the forces and torques involved in manipulator arm movements. Computed Torque Control involves calculating the required torques to achieve desired motions, compensating for dynamic effects. Impedance Control allows manipulators to adapt to external forces, providing flexibility in dynamic environments. Hybrid Position/Force Control combines position and force control strategies, offering precise control in tasks requiring both positional accuracy and force application.

1.5.3 Motion Planning

Motion planning for manipulator arms involves generating collision-free trajectories. Algorithms such as Probabilistic Roadmap (PRM) and Rapidly-exploring Random Trees (RRT) are used to explore feasible paths and avoid obstacles. These techniques ensure that manipulator arms can move effectively and safely within their workspace.

1.5.4 Safety and Human-Robot Interaction

Safety and human-robot interaction are critical aspects of manipulator arm operation. Force/torque limits prevent excessive forces that could damage the robot or its environment. Collision detection techniques identify potential collisions and allow for real-time adjustments. Human-Robot Interaction (HRI) frameworks are designed to enhance collaboration between humans and robots, focusing on safety and ergonomic considerations.

1.6 Command and Control Strategies for Mobile Manipulator Robots

1.6.1 Integration of Mobile and Manipulator Systems

Integrating mobile bases with manipulator arms presents unique challenges, such as coordinating movements and managing complex interactions between mobility and manipulation. Coordinated control strategies ensure that mobile manipulators operate effectively, balancing mobility with manipulation tasks.

1.6.2 Motion Planning in Dynamic Environments

Motion planning for mobile manipulators in dynamic environments involves adapting to changing conditions and obstacles. Real-time obstacle avoidance and path replanning techniques are employed to ensure safe and efficient operation.

1.6.3 Applications of Mobile Manipulators

Mobile manipulators have diverse applications in industries, services, and research. Case studies illustrate their use in tasks such as automated material handling, service robotics, and scientific research. Future trends and challenges include enhancing capabilities, improving integration, and addressing complex operational scenarios.

1.7 Programming and Software Tools

1.7.1 Programming Robots in C, Python, and MATLAB

The programming of robotic systems often involves a combination of C++, Python, and MATLAB, each contributing distinct advantages to the development process. C++ is renowned for its high performance and efficiency, making it ideal for real-time control and systems requiring fine-grained hardware access. Its low-level programming capabilities facilitate precise management of resources and high-speed execution, crucial for performance-critical applications. In contrast, Python excels in rapid development and prototyping due to its simplicity and extensive libraries. It offers a high level of abstraction, allowing for quick development of algorithms and integration with various frameworks, such as OpenCV for computer vision. Pythons ease of use and readability also accelerate the development cycle and facilitate testing and debugging. MATLAB complements these languages with its robust mathematical and computational tools, providing a powerful environment for algorithm development and data analysis. MATLABs ability to handle complex mathematical operations and its integrated development environment support rapid prototyping and optimization of robotic algorithms. By leveraging the strengths of C++, Python, and MATLAB together, developers can achieve a balanced approach, combining efficient real-time performance with rapid development and advanced mathematical analysis to create well-rounded and optimized robotic systems.

Figure 1.1: Robotic System Programming and Development Workspace

1.7.2 Robot Operating Systems (ROS)

Robot Operating System (ROS) is a crucial framework in modern robotics, offering a robust infrastructure for developing and managing robotic software. ROS simplifies the integration of various software components by providing a standardized framework for communication. It organizes robotic systems into modular units known as nodes, which communicate via topics and services. Topics enable nodes to publish and subscribe to data streams asynchronously, while services facilitate synchronous communication through request-response interactions. This modularity and communication efficiency streamline the development of complex robotic applications, allowing for scalable and maintainable software architectures.

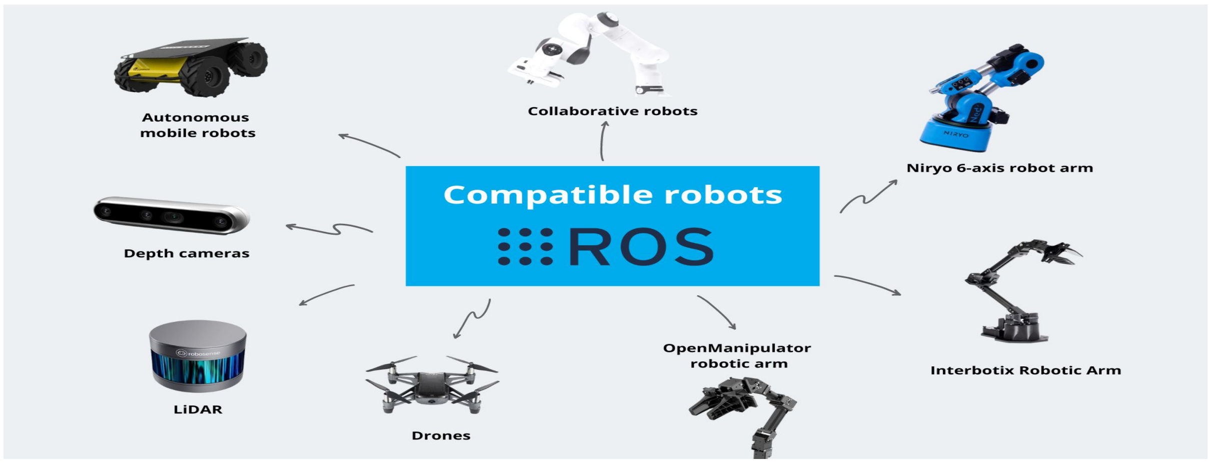


Figure 1.2: ROS-Compatible Robots and Devices

1.7.3 Simulation and Testing

Simulation and testing are indispensable in the development of robotic systems, offering tools to validate designs and algorithms before physical deployment. Gazebo and MATLAB/Simulink are prominent simulation platforms that provide realistic environments for testing robotic systems. Gazebo integrates with ROS to simulate dynamic interactions with the environment, while MATLAB/Simulink offers a comprehensive suite for modeling, simulating, and analyzing robotic systems. These tools enable engineers to perform extensive testing under various scenarios, mitigating risks and reducing costs associated with physical prototyping. The ability to simulate complex interactions and conditions is crucial for optimizing robotic performance and ensuring reliability.

1.7.4 Real-time Programming

Real-time programming is essential for applications requiring precise and timely control, such as robotics. Ensuring real-time performance involves addressing challenges related to concurrency and timing. Techniques such as task scheduling, interrupt handling, and real-time operating systems (RTOS) are employed to manage the execution of concurrent tasks and maintain stringent timing constraints. Effective real-time programming is critical for applications where delays or jitter can impact system performance, such as in autonomous navigation or real-time control of robotic arms.

1.8 Electronic Components in Command and Control Systems

1.8.1 Sensors and Actuators

Sensors and actuators form the core of robotic systems, providing essential data and performing actions based on control commands. Various types of sensors, including encoders, Inertial Measurement Units (IMUs), cameras, and Light Detection and Ranging (LIDAR) systems, are utilized to perceive the robot's environment and its own state. Encoders measure rotational positions and velocities, IMUs provide orientation and acceleration data, cameras capture visual information, and LIDAR systems enable detailed distance measurements. Actuators, including DC motors, servo motors, stepper motors, and hydraulic actuators, execute movements and control mechanisms. Each actuator type offers different capabilities in terms of precision, range, and force, making them suitable for diverse applications in robotics.

1.8.2 Power Management in Robotic Systems

Power management is a critical aspect of robotic system design, influencing both performance and operational efficiency. The selection of power supplies for mobile robots and manipulators must account for factors such as voltage, current, and battery life. Battery management systems play a pivotal role in monitoring and optimizing battery performance, ensuring that energy consumption is efficiently managed. Energy-efficient design strategies, including the use of low-power components and regenerative braking systems, contribute to extending operational time and reducing overall energy consumption.

1.8.3 Interface Electronics and Signal Conditioning

Interface electronics and signal conditioning are fundamental for ensuring accurate communication between sensors, actuators, and control systems. Signal conditioning circuits are employed to enhance sensor signals, improving accuracy and reliability. These circuits may include amplification, filtering, and analog-to-digital conversion (ADC) components. Communication interfaces, such as digital-to-analog converters (DACs) and various communication protocols, facilitate data exchange between electronic components and control systems. Proper design and implementation of these interfaces are essential for maintaining signal integrity and ensuring seamless integration of various system components.

1.9 Advantages of Command and Control in Mobile Manipulator Robots in Industry

The application of command and control systems in mobile manipulator robots offers numerous advantages within industrial settings. These robots combine mobility with versatile manipulation capabilities, enhancing their utility across a range of tasks. The integration of advanced command and control systems enables precise and adaptive operation, allowing for dynamic adjustments in response to varying conditions and tasks. This adaptability contributes to increased productivity and efficiency, as robots can perform complex tasks with minimal human intervention. Moreover, the ability to integrate various sensors and actuators within a unified control framework enhances the robot's capability to interact with its environment and execute intricate operations. The deployment of mobile manipulator robots equipped with sophisticated command and control systems not only improves operational flexibility but also reduces the need for manual labor, leading to significant cost savings and enhanced safety in industrial environments.

Chapter 2

Modeling

Modeling in robotics is essential for developing and operating robotic systems. It involves creating mathematical representations of a robot's structure and behavior, which helps predict its motion and interactions with the environment. Effective modeling allows engineers to analyze designs, implement control algorithms, and ensure reliable performance in real-world applications. As technology advances, models have become more complex, requiring sophisticated techniques to handle various operational scenarios.
In this chapter, we focus on modeling mobile manipulator robots, particularly their kinematics. Kinematics examines the geometric and spatial relationships between different robot components without considering the forces involved (dynamics). By studying kinematics, we can understand how these robots move and interact with their surroundings. This knowledge is crucial for designing, controlling, and optimizing mobile manipulator systems.

2.1 Robot Mobile

The kinematics of a four-wheeled robot equipped with Mecanum wheels involves a unique design that facilitates omnidirectional movement. This robot is positioned on two parallel axles, with its center of mass denoted as point C. Each Mecanum wheel allows for lateral movement, thanks to its angled rollers, enhancing the robot's maneuverability. The distance from the robot's center of mass C to each wheel axle is represented by ρ, while the spacing between the centers of the wheels is $2l$. In a fixed coordinate system XOY, the robot's position is defined by the coordinates x_c and y_c. Additionally, the angle ψ describes the orientation of the robot's longitudinal axis in relation to the OX axis. The wheels rotate at angles φ_i, which are measured relative to axes that are perpendicular to their respective planes and intersect at their centers Figure (2.1). This configuration enables precise control over both translational and rotational movements, allowing the robot to navigate through complex environments with agility and precision [8].

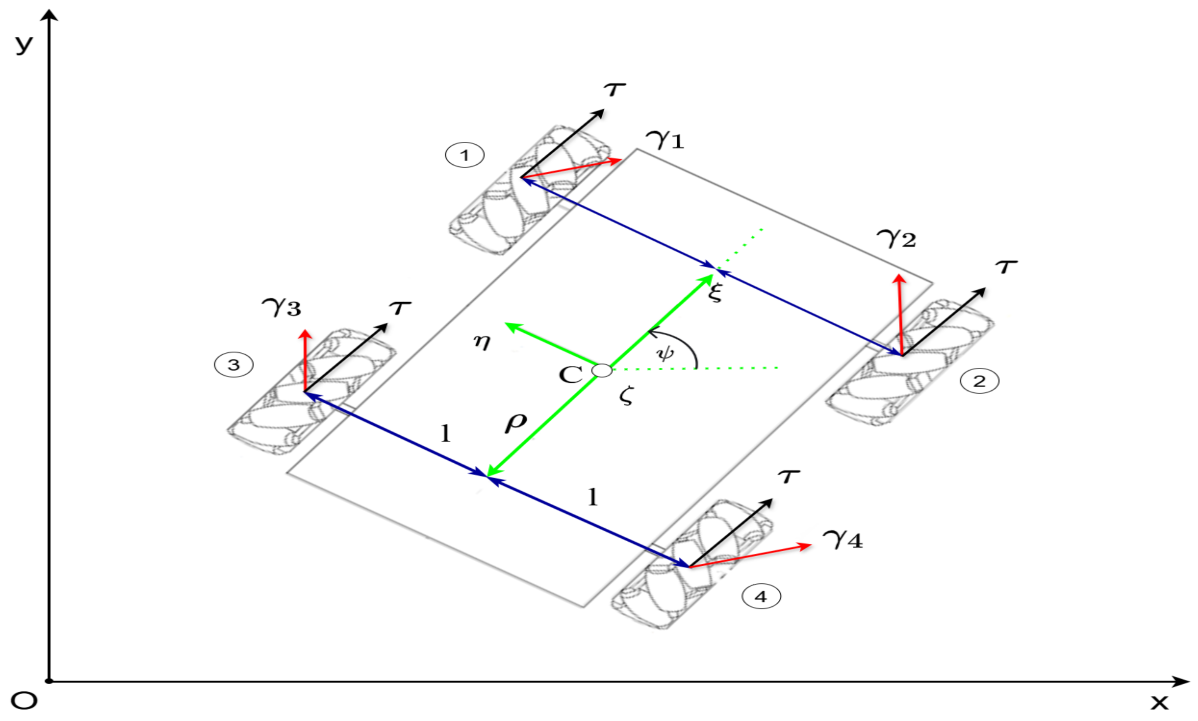


Figure 2.1: Four wheeled mobile robot with Mecanum wheels

2.1.1 Model of a Mecanum Wheel

A Mecanum wheel is an omni-directional wheel designed to enable a vehicle to move in any direction by combining the motion from multiple wheels Figure (2.2). This wheel is modeled as a thin disk of radius R, which assumes that the rollers, fixed around the circumference of the wheel, are small compared to the wheels diameter. Each roller is mounted at an angle δ with respect to the plane of the wheel, typically $\delta = 45°$. This configuration allows the wheel to roll in various directions while minimizing slip.

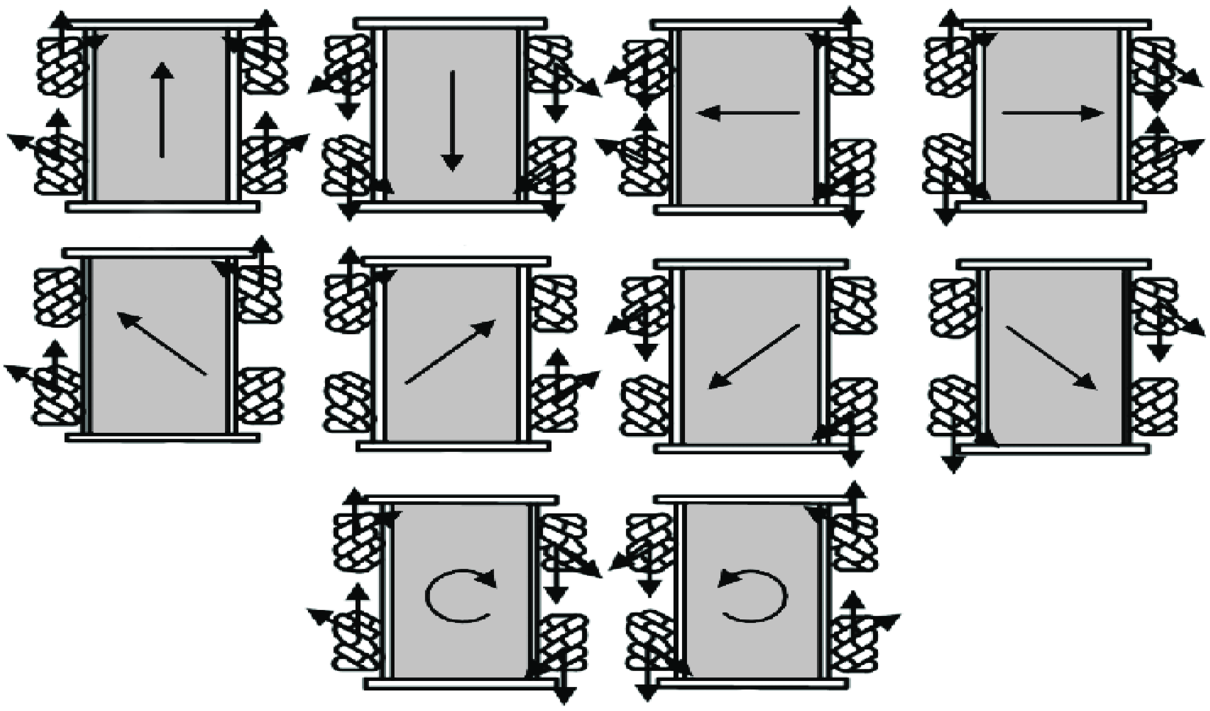


Figure 2.2: Mecanum Wheel Omni-Directional Movement

To model the Mecanum wheel's behavior, we define several key variables: $\mathbf{V}_P$ as the velocity of the contact point P with the ground, $\mathbf{V}_K$ as the velocity of the wheels center K, $\boldsymbol{\omega}$ as the angular velocity of the wheel, and r as the vector from the center K to the contact point P Figure(2.3).

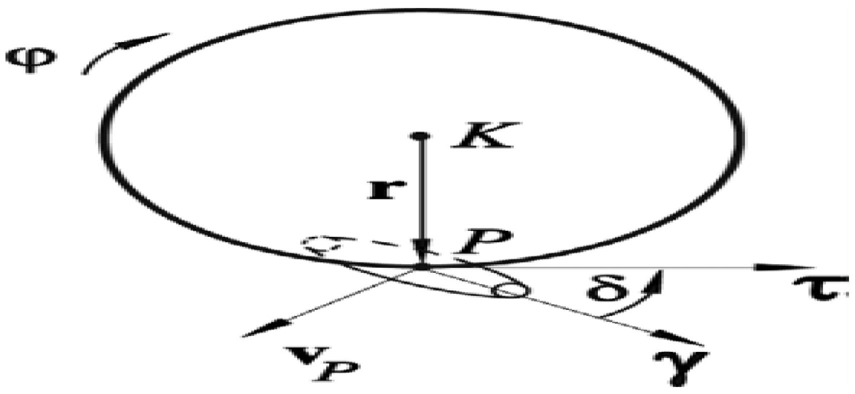


Figure 2.3: Model of a Mecanum Wheel

The velocity of point P can be expressed as the sum of the translational velocity of the wheels center and the rotational contribution, i.e.,

$$\mathbf{V}_P = \mathbf{V}_K + \boldsymbol{\omega} \times \mathbf{r}. \tag{2.1}$$

Since the wheel operates without slip, the velocity at the contact point must be orthogonal to the axis of the rollers. Let $\boldsymbol{\gamma}$ be the unit vector along the rollers axis. The no-slip condition is therefore given by

$$\mathbf{V}_P \cdot \boldsymbol{\gamma} = 0, \tag{2.2}$$

implying that the component of the velocity $\mathbf{V}_P$ along $\boldsymbol{\gamma}$ is zero. Substituting $\mathbf{V}_P$ into this condition results in

$$(\mathbf{V}_K + \boldsymbol{\omega} \times \mathbf{r}) \cdot \boldsymbol{\gamma} = 0. \tag{2.3}$$

Expanding this expression, we get

$$\mathbf{V}_K \cdot \boldsymbol{\gamma} + (\boldsymbol{\omega} \times \mathbf{r}) \cdot \boldsymbol{\gamma} = 0. \tag{2.4}$$

The term $(\boldsymbol{\omega} \times \mathbf{r}) \cdot \boldsymbol{\gamma}$ can be simplified considering that $\boldsymbol{\gamma}$ is perpendicular to the tangential direction of the wheel. Given the angle δ between the roller axis $\boldsymbol{\gamma}$ and the radial direction $\mathbf{r}$, and assuming that the angular velocity $\boldsymbol{\omega}$ is aligned along the axis perpendicular to the wheels plane, the magnitude of the component of $\mathbf{V}_K$ along $\boldsymbol{\gamma}$ can be related to the angular velocity $\dot{\varphi}$ of the wheel.
Finally, substituting the contribution of $\boldsymbol{\omega} \times \mathbf{r}$, which is $-R\dot{\varphi}\cos\delta$, into the no-slip condition yields the final expression

$$\mathbf{V}_K \cdot \boldsymbol{\gamma} = R\dot{\varphi}\cos\delta \tag{2.5}$$

This equation establishes a direct relationship between the velocity of the wheels center, the wheels angular velocity, and the angle δ between the roller axis and the plane of the wheel.

2.1.2 Kinematic Constraint Equations

In the context of analyzing the kinematics of a robot with Mecanum wheels, we start with the fundamental kinematic constraint that describes the rolling without slipping condition. This constraint is initially defined by Equation (2.5).
For simplification, we assume that the angle δ is set to $\frac{\pi}{4}$. This is a common assumption for Mecanum wheels, where the rollers are typically oriented at 45° to the wheel axis.Thus, the equation reduces to:

$$\mathbf{V}_K \cdot \boldsymbol{\gamma} = \frac{R}{\sqrt{2}}\dot{\varphi} \tag{2.6}$$

Let $\mathbf{V}_C$ denote the velocity of the center of mass of the robot, and let $\mathbf{r}_i$ be the vector from the center of mass to the center of the i-th wheel. The velocity of the wheel is given by:

$$\mathbf{V}_{K_i} = \mathbf{V}_C + \Omega \times \mathbf{r}_i \quad i = 1, \ldots, 4, \tag{2.7}$$

where Ω is the angular velocity of the robot. Substituting $\mathbf{V}_{K_i}$ into the kinematic constraint equation, we obtain:

$$(\mathbf{V}_C + \Omega \times \mathbf{r}_i) \cdot \gamma_i = \frac{R}{\sqrt{2}}\dot{\varphi}_i \quad i = 1, \ldots, 4, \tag{2.8}$$

Expanding $\Omega \times \mathbf{r}_i$ as $(\mathbf{r}_i \times \gamma_i) \cdot \Omega$, we rewrite the constraint as:

$$\mathbf{V}_C \cdot \gamma_i + (\mathbf{r}_i \times \gamma_i) \cdot \Omega = \frac{R}{\sqrt{2}}\dot{\varphi}_i \quad i = 1, \ldots, 4, \tag{2.9}$$

Introduce a robot-attached coordinate system $\xi\eta\zeta$, where the ξ axis is aligned with the longitudinal axis, η with the lateral axis, and ζ is vertical. Denote $V_{C\xi}$ and $V_{C\eta}$ as the projections of the velocity of the center of mass onto the ξ and η axes, respectively. Substituting into the constraint equations, we have:

$$V_{C\xi}\gamma_{i\xi} + V_{C\eta}\gamma_{i\eta} + (r_{i\xi}\gamma_{i\eta} - r_{i\eta}\gamma_{i\xi})\dot{\psi} = \frac{R}{\sqrt{2}}\dot{\varphi}_i \quad i = 1, \ldots, 4, \tag{2.10}$$

For Mecanum wheels, the components are:

$$\begin{aligned} \gamma_{1\xi} &= \gamma_{4\xi} = \frac{1}{\sqrt{2}}, & \gamma_{1\eta} &= \gamma_{4\eta} = -\frac{1}{\sqrt{2}} \\ \gamma_{2\xi} &= \gamma_{3\xi} = \frac{1}{\sqrt{2}}, & \gamma_{2\eta} &= \gamma_{3\eta} = \frac{1}{\sqrt{2}} \\ r_{1\xi} &= r_{2\xi} = \rho, & r_{1\eta} &= r_{3\eta} = l \\ r_{3\xi} &= r_{4\xi} = -\rho, & r_{2\eta} &= r_{4\eta} = -l \end{aligned} \tag{2.11}$$

Substituting these values into the constraint equations yields:

$$\begin{aligned} V_{C\xi} - V_{C\eta} - (\rho + l)\dot{\psi} &= R\dot{\varphi}_1 \\ V_{C\xi} + V_{C\eta} + (\rho + l)\dot{\psi} &= R\dot{\varphi}_2 \\ V_{C\xi} + V_{C\eta} - (\rho + l)\dot{\psi} &= R\dot{\varphi}_3 \\ V_{C\xi} - V_{C\eta} + (\rho + l)\dot{\psi} &= R\dot{\varphi}_4 \end{aligned} \tag{2.12}$$

Solving these for $V_{C\xi}$, $V_{C\eta}$, and $\dot{\psi}$ gives:

$$V_{C\xi} = \frac{R}{2}(\dot{\varphi}_1 + \dot{\varphi}_2) \tag{2.13}$$

$$V_{C\eta} = \frac{R}{2}(\dot{\varphi}_3 - \dot{\varphi}_1) \tag{2.14}$$

$$\dot{\psi} = \frac{R}{2(\rho + l)}(\dot{\varphi}_2 - \dot{\varphi}_3) \tag{2.15}$$

$$\dot{\varphi}_1 + \dot{\varphi}_2 = \dot{\varphi}_3 + \dot{\varphi}_4 \tag{2.16}$$

The velocities of the center of mass in the fixed reference frame are related to $V_{C\xi}$ and $V_{C\eta}$ as follows:

$$V_{C\xi} = \dot{x}_c \cos\psi + \dot{y}_c \sin\psi \tag{2.17}$$

$$V_{C\eta} = -\dot{x}_c \sin\psi + \dot{y}_c \cos\psi \tag{2.18}$$

which can be inverted to:

$$\dot{x}_c = V_{C\xi} \cos\psi - V_{C\eta} \sin\psi \tag{2.19}$$

$$\dot{y}_c = V_{C\xi} \sin\psi + V_{C\eta} \cos\psi \tag{2.20}$$

Substituting $V_{C\xi}$ and $V_{C\eta}$ from Equation (2.13) and (2.14) into the expressions for $\dot{x}_c$ and $\dot{y}_c$, we obtain:

$$\dot{x}_c = \frac{R}{\sqrt{2}} \cos\left(\psi - \frac{\pi}{4}\right) \dot{\varphi}_1 + \frac{R}{2}(\cos\psi\dot{\varphi}_2 - \sin\psi\dot{\varphi}_3) \tag{2.21}$$

$$\dot{y}_c = \frac{R}{\sqrt{2}} \sin\left(\psi - \frac{\pi}{4}\right) \dot{\varphi}_1 + \frac{R}{2}(\sin\psi\dot{\varphi}_2 + \cos\psi\dot{\varphi}_3) \tag{2.22}$$

Integrating the remaining equations from Equation (2.15) and (2.16), we have:

$$\psi = \frac{R}{2(\rho + l)}(\dot{\varphi}_2 - \dot{\varphi}_3) + C_1 \tag{2.23}$$

$$\varphi_4 = \varphi_1 + \varphi_2 - \varphi_3 + C_2 \tag{2.24}$$

where C_1 and C_2 are integration constants. Thus, the system exhibits two holonomic constraints (constraints that can be expressed in terms of the system's configuration variables), allowing us to eliminate the generalized coordinates ψ and φ_4. Using these constraints, the velocities of the center of mass can be expressed as:

$$\dot{x}_c = a_1\dot{\varphi}_1 + a_2\dot{\varphi}_2 + a_3\dot{\varphi}_3 \tag{2.25}$$

$$\dot{y}_c = b_1\dot{\varphi}_1 + b_2\dot{\varphi}_2 + b_3\dot{\varphi}_3 \tag{2.26}$$

where the coefficients are:

$$\begin{aligned} a_1 &= \frac{R}{\sqrt{2}} \cos\left(\psi - \frac{\pi}{4}\right), & b_1 &= \frac{R}{\sqrt{2}} \sin\left(\psi - \frac{\pi}{4}\right) \\ a_2 &= \frac{R}{2} \cos\psi, & b_2 &= \frac{R}{2} \sin\psi \\ a_3 &= -\frac{R}{2} \sin\psi, & b_3 &= \frac{R}{2} \cos\psi \end{aligned} \tag{2.27}$$

Skew-Symmetric Quantities

Skew-symmetric quantities are used to analyze non-holonomic constraints in mechanical systems. These constraints describe the relationship between the system's velocities and generalized coordinates, helping determine whether the constraints are integrable (holonomic) or not (non-holonomic).

Calculation of α_{ij}

The skew-symmetric quantities α_{ij} are computed from the coefficients a_i, which relate Cartesian velocities to generalized velocities. The formula for α_{ij} is:

$$\alpha_{ij} = \frac{\partial a_i}{\partial \varphi_j} - \frac{\partial a_j}{\partial \varphi_i}$$

Here, $\frac{\partial a_i}{\partial \varphi_j}$ represents the partial derivative of the coefficient a_i with respect to the generalized coordinate φ_j, and $\frac{\partial a_j}{\partial \varphi_i}$ is the partial derivative of the coefficient a_j with respect to φ_i. The skew-symmetric property ensures that $\alpha_{ij} = -\alpha_{ji}$.

Calculation of β_{ij}

Similarly, the skew-symmetric quantities β_{ij} are derived from the coefficients b_i. The formula for β_{ij} is:

$$\beta_{ij} = \frac{\partial b_i}{\partial \varphi_j} - \frac{\partial b_j}{\partial \varphi_i}$$

In this case, $\frac{\partial b_i}{\partial \varphi_j}$ is the partial derivative of the coefficient b_i with respect to φ_j, and $\frac{\partial b_j}{\partial \varphi_i}$ is the partial derivative of b_j with respect to φ_i. The skew-symmetric property here ensures that $\beta_{ij} = -\beta_{ji}$.

The presence of non-zero values in α_{ij} and β_{ij} indicates non-holonomic constraints, which impose restrictions on velocities rather than just positions. Understanding and calculating these skew-symmetric quantities are crucial for analyzing the dynamics and constraints of mechanical systems.

To confirm that these constraints are non-holonomic (constraints that are dependent on velocities and cannot be integrated into positional constraints), we calculate the non-zero skew-symmetric quantities α_{ij} and β_{ij}, which are given by:

$$\begin{aligned}
\alpha_{12} &= -\frac{R^2}{2\sqrt{2}} \sin\left(\psi - \frac{\pi}{4}\right) & \beta_{12} &= \frac{R^2}{2\sqrt{2}} \sin\left(\psi + \frac{\pi}{4}\right) \\
\alpha_{13} &= \frac{R^2}{2\sqrt{2}} \sin\left(\psi - \frac{\pi}{4}\right) & \beta_{13} &= -\frac{R^2}{2\sqrt{2}} \sin\left(\psi + \frac{\pi}{4}\right) \\
\alpha_{23} &= \frac{R^2}{2\sqrt{2}} \sin\left(\psi + \frac{\pi}{4}\right) & \beta_{23} &= \frac{R^2}{2\sqrt{2}} \sin\left(\psi - \frac{\pi}{4}\right)
\end{aligned} \tag{2.28}$$

2.1.3 Techniques for Approximating Equations of Motion

In analyzing the kinematic constraints and deriving the equations of motion for a robot equipped with Mecanum wheels, we start by considering the system of linear equations that relate the wheel velocities $\dot{\varphi}_i$ to the robot's chassis velocities $V_{C\xi}, V_{C\eta}$ and angular velocity $\dot{\psi}$. The kinematic constraints can be expressed as

$$\dot{\varphi} = JV, \tag{2.29}$$

where:

- $\dot{\varphi}$ is a 4×1 vector of wheel velocities,

- J is a 4×3 matrix,
- V is a 3×1 vector of chassis velocities.

Specifically, the vectors and matrix are defined as follows:

$$\dot{\varphi} = \begin{pmatrix} \dot{\varphi}_1 \\ \dot{\varphi}_2 \\ \dot{\varphi}_3 \\ \dot{\varphi}_4 \end{pmatrix}, \quad V = \begin{pmatrix} V_{C\xi} \\ V_{C\eta} \\ (\rho + l)\dot{\psi} \end{pmatrix}, \quad J = \frac{1}{R} \begin{pmatrix} 1 & -1 & -1 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \end{pmatrix}. \tag{2.30}$$

Since the system is overdetermined with four equations for three unknowns, it does not have a solution for arbitrary values of $\dot{\varphi}_1, \dot{\varphi}_2, \dot{\varphi}_3$, and $\dot{\varphi}_4$. For the system to have a solution, the equations must be linearly dependent, leading to the compatibility condition:

$$\dot{\varphi}_1 + \dot{\varphi}_2 = \dot{\varphi}_3 + \dot{\varphi}_4. \tag{2.31}$$

This ensures that the wheel velocities are consistent and do not conflict with each other. To find an approximate solution, we use the pseudoinverse of the matrix J. Premultiplying the original equation by the transpose J^T gives:

$$J^T \dot{\varphi} = J^T J V. \tag{2.32}$$

The matrix $J^T J$ is invertible, and its inverse is used to solve for V:

$$V = (J^T J)^{-1} J^T \dot{\varphi}. \tag{2.33}$$

Here, $J^+ = (J^T J)^{-1} J^T$ is the pseudoinverse of J. The specific form of the pseudoinverse for our matrix J is:

$$J^+ = \frac{R}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{pmatrix}. \tag{2.34}$$

Using the pseudoinverse, we find the chassis velocities as follows:

$$\begin{aligned} V_{C\xi} &= \frac{R}{4}(\dot{\varphi}_1 + \dot{\varphi}_2 + \dot{\varphi}_3 + \dot{\varphi}_4), \\ V_{C\eta} &= \frac{R}{4}(-\dot{\varphi}_1 + \dot{\varphi}_2 + \dot{\varphi}_3 - \dot{\varphi}_4), \\ \dot{\psi} &= \frac{R}{4(\rho + l)}(-\dot{\varphi}_1 + \dot{\varphi}_2 - \dot{\varphi}_3 + \dot{\varphi}_4). \end{aligned} \tag{2.35}$$

These equations provide a solution that minimizes the sum of the squared discrepancies between the left and right sides of the original kinematic equations.
If the compatibility condition Equation (2.1.3) is satisfied, the exact solution to the

kinematic equations can be expressed as:

$$\begin{aligned}
V_{C\xi} &= \frac{R}{2}(\dot{\varphi}_1 + \dot{\varphi}_2), \\
V_{C\eta} &= \frac{R}{4}(-\dot{\varphi}_1 + \dot{\varphi}_2 + \dot{\varphi}_3 - \dot{\varphi}_4), \\
\dot{\psi} &= \frac{R}{2(\rho + l)}(\dot{\varphi}_2 - \dot{\varphi}_3).
\end{aligned} \tag{2.36}$$

This approach of using the pseudoinverse to solve an overdetermined system is widely adopted in robotics and other fields, as it provides a way to find the best possible solution when an exact solution is not feasible due to more constraints than degrees of freedom. The pseudoinverse method ensures that the calculated chassis velocities $V_{C\xi}, V_{C\eta}$ and $\dot{\psi}$ adhere to the kinematic constraints imposed by the Mecanum wheels, ensuring accurate and effective control of the robot's motion.

2.2 Robotic Arm

The first phase is that of modeling the robot on which the project is based, so as to get the Denavit Hartenberg parameter table.
We are using robot arm that is depicted in Figure (2.4).
This arm has a similar kinematic configuration as some of the industrial manipulators.
The Arm is made out of plastic and has six servo motors as actuators.
This robot contains six main parts:

- The base that enables rotation of the robot around vertical axis
- The shoulder is the second link of manipulator, it controls vertical and forward/backward movements of manipulator
- The elbow controls the movement in the same axes as shoulder
- The wrist vertical movement controls arms attack angle at which are objects picked up,
- The wrist rotation that handles the rotation of the gripper around the X axis
- The gripper is the part of arm that interacts with its surroundings this particular one has jaws for picking and moving objects

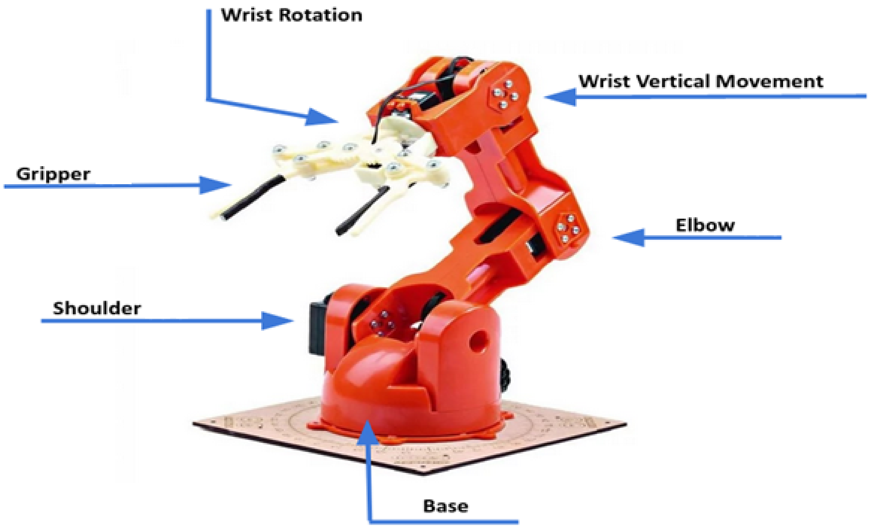


Figure 2.4: Main Components of the Robot Manipulator

The robot manipulator is a functional robotic arm that can be controlled using Arduino technology. It is versatile and can be configured in various ways to perform different tasks, including object manipulation. Users have the flexibility to assemble the robot in numerous configurations Figure (2.5).
Here are a few examples:

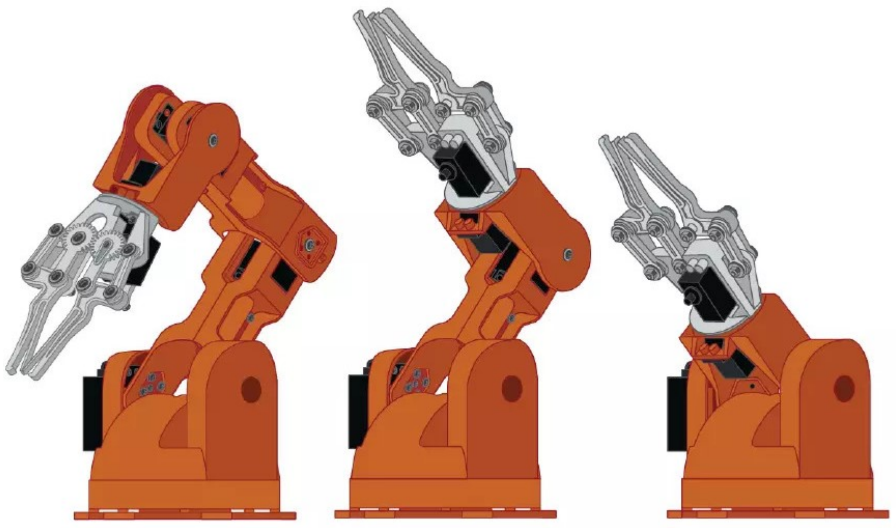


Figure 2.5: Robot Manipulator with Multitude Ways

The robot manipulator can also carry various objects on the end of its arm Figure (2.6). For example, the robot can be used:

- With a camera to follow a subject,
- By setting up a phone or tablet to track movement during a video conference,
- With a solar panel to follow the sun.

Figure 2.6: Robot arm with different object

2.2.1 Denavit-Hartenberg parameters

Denavit-Hartenberg (DH) parameters, also referred to as Denavit-Hartenberg parameters, consist of four parameters that are essential in mechanical engineering for defining a specific convention for linking reference frames to the links of a spatial kinematic chain or robot manipulator. This convention was introduced by Jacques Denavit and Richard Hartenberg in 1955 [9].
The DH convention defines the following parameters:

- θ_i: The rotation around the z-axis.
- d_i: The sliding motion along the z-axis.
- α_i: The angle measured around the X axis.
- r_i: The distance measured along the X axis.

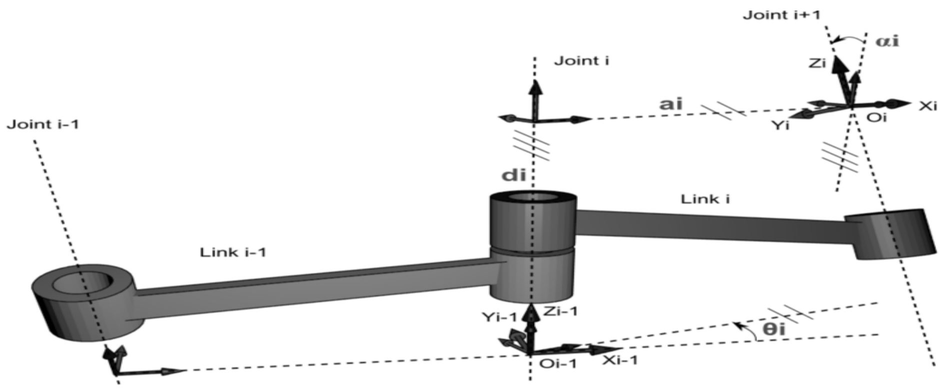


Figure 2.7: Standard DH Parameters

The Denavit-Hartenberg (D-H) convention provides a systematic method for representing the geometric configuration of robotic arms. By defining the relationship between adjacent links through a set of parameters, namely the link length, link twist, link offset, and joint angle, we can effectively describe the kinematic structure of a robotic manipulator. The table (2.1) outlines the D-H parameters for the robotic arm under consideration, facilitating the analysis of its movement and the computation of its forward and inverse kinematics.

Joint	α (°)	d (mm)	a (mm)	θ (°)
1	-90	L0+L1=70	0	θ1
2	0	0	L2=-125	θ2+90
3	0	0	L3=-125	θ3
4	90	0	0	θ4-90
5	0	L4+L5=190	0	θ5

Table 2.1: D-H parameters table

2.3 Kinematic Modeling of Manipulator arm

Forward and inverse kinematics are fundamental concepts in robotics, integral to controlling and predicting the movements of robotic systems. Forward kinematics involves determining the position and orientation of a robot's end-effector (such as a hand or tool) based on given joint parameters, such as angles or displacements. This process is relatively straightforward, relying on the known configuration of the robot's joints and links to compute the resulting spatial position through a series of transformations.
In contrast, inverse kinematics is more complex and involves calculating the necessary joint parameters to achieve a desired position and orientation of the end-effector. This requires solving potentially nonlinear equations and often necessitates iterative methods or optimization techniques due to the possibility of multiple or no solutions [10]. Both forward and inverse kinematics are crucial for precise robotic motion control, enabling tasks ranging from simple pick-and-place operations to complex manipulations in dynamic environments.

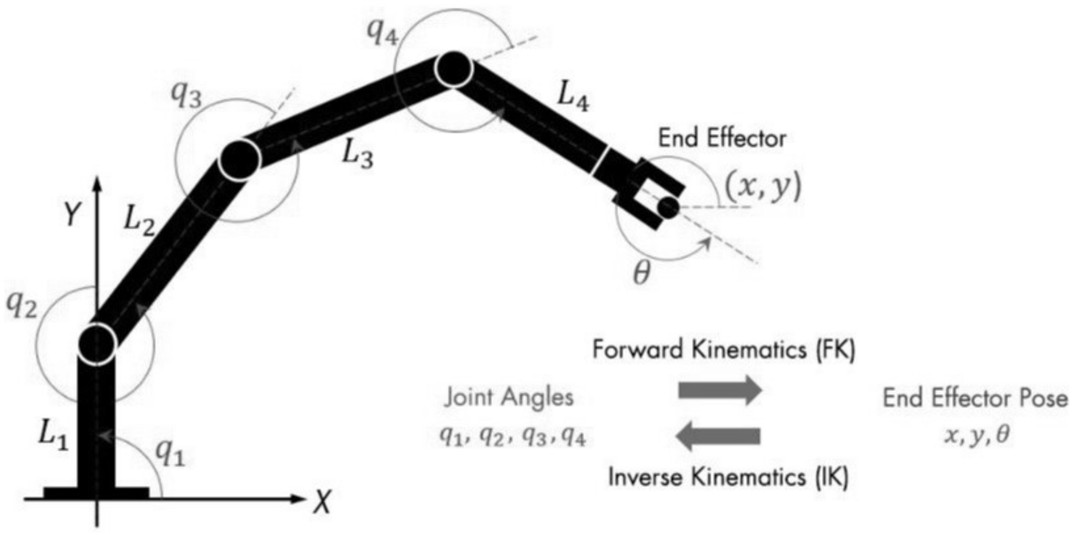


Figure 2.8: Joint Position Configuration in Robotic Manipulation

Table (2.2) compares forward and inverse kinematics in robotics. Forward kinematics involves calculating the position and orientation of the end-effector based on given joint variables (θ or d). In contrast, inverse kinematics requires determining the joint variables needed to achieve a desired position and orientation of the end-effector.

	Forward Kinematics	Inverse Kinematics
Given	Joint Variables q (θ or d)	Position and orientation of end-effector, p
Required	Position and orientation of end-effector, p	Joint Variables q (θ or d) to get p
	p = f(q_1, q_2, . . . , q_n) = f(q)	q = f(p)

Table 2.2: Comparison of Forward and Inverse Kinematics in Robotics

2.3.1 Forward Kinematics

Forward kinematics is a crucial technique in robotics and computer graphics used to determine the position and orientation of a robot's end effector or a character's limb based on given joint parameters. By knowing the lengths and angles of each joint and link in a kinematic chain, forward kinematics employs mathematical models to calculate the final position of each segment. This process is essential for ensuring accurate and predictable movements in robotic systems, enabling tasks such as picking and placing objects, navigating through spaces, and performing complex manipulations. Understanding forward kinematics is fundamental for designing and controlling robotic mechanisms and animated figures [1].

The process involves defining transformation matrices for each joint ($[X_i]$) and link ($[Z_i]$) along the robotic arm. The joint transformation matrices account for both rotational and translational motion at each joint, while the link transformation matrices represent the translation along each link. By multiplying these matrices in the proper order, from the base to the end effector, we obtain the overall transformation matrix ($[A]$), which encapsulates the cumulative effect of all joint motions and link translations. This equation,

$$[A] = [Z_1][X_1][Z_2][X_2]\dots[X_{n-1}][Z_n], \tag{2.37}$$

provides a concise way to express the relationship between joint variables and the end effector's position and orientation.
To achieve this, we use a series of transformations that describe how each link in the robot relates to its previous link. These transformations are typically represented using homogeneous transformation matrices. The link transformation from frame $i-1$ to frame i, denoted as ${}^{i-1}A_i$, can be constructed using four basic transformations: translation along the Z-axis by d_i, rotation around the Z-axis by θ_i, translation along the X-axis by a_i, and rotation around the X-axis by α_i. This can be expressed as:

$${}^{i-1}A_i = \mathrm{Trans}_{Z_i}(d_i)\,\mathrm{Rot}_{Z_i}(\theta_i)\,\mathrm{Trans}_{X_i}(a_i)\,\mathrm{Rot}_{X_i}(\alpha_i), \tag{2.38}$$

where $\mathrm{Trans}_{Z_i}(d_i)$ represents the translation along the Z-axis by d_i,

$$\mathrm{Trans}_{Z_i}(d_i) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.39}$$

$\mathrm{Rot}_{Z_i}(\theta_i)$ represents the rotation around the Z-axis by θ_i,

$$\mathrm{Rot}_{Z_i}(\theta_i) = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.40}$$

$\mathrm{Trans}_{X_i}(a_i)$ represents the translation along the X-axis by a_i,

$$\mathrm{Trans}_{X_i}(a_i) = \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.41}$$

and $\mathrm{Rot}_{X_i}(\alpha_i)$ represents the rotation around the X-axis by α_i,

$$\mathrm{Rot}_{X_i}(\alpha_i) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.42}$$

Combining these four transformations into a single homogeneous transformation matrix ${}^{i-1}A_i$, we get:

$$
{}^{i-1}A_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.43}
$$

This matrix ${}^{i-1}A_i$ describes the position and orientation of link i with respect to link $i-1$. By sequentially multiplying these matrices from the base to the end-effector, the overall transformation matrix is obtained, which describes the end-effector's position and orientation relative to the base frame.
The final product of this procedure for robot manipulator can be seen in Figure (2.9)

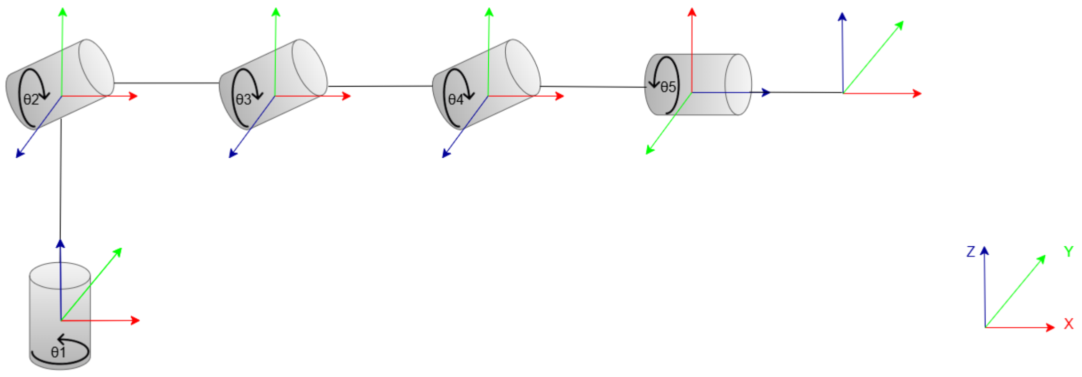


Figure 2.9: Geometric Representation of the Robotic Arm for Inverse Kinematics

Homogeneous transformation matrices for each robot link are determined using Denavit-Hartenberg (D-H) parameters, as specified in Equation (2.43). These matrices represent the relationship between consecutive links and are computedfor accurate position and orientation modeling.
For Link 1, the transformation matrix is given by:

$$
A_1 = \begin{bmatrix} \cos(\theta_1) & 0 & -\sin(\theta_1) & 0 \\ \sin(\theta_1) & 0 & \cos(\theta_1) & 0 \\ 0 & -1 & 0 & 70 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.44}
$$

For Link 2, the transformation matrix is given by:

$$
A_2 = \begin{bmatrix} \cos(\theta_2+\frac{\pi}{2}) & -\sin(\theta_2+\frac{\pi}{2}) & 0 & -125\cos(\theta_2+\frac{\pi}{2}) \\ \sin(\theta_2+\frac{\pi}{2}) & \cos(\theta_2+\frac{\pi}{2}) & 0 & -125\sin(\theta_2+\frac{\pi}{2}) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.45}
$$

For Link 3, the transformation matrix is given by:

$$A_3 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & -125\cos(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & -125\sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.46}$$

For Link 4, the transformation matrix is given by:

$$A_4 = \begin{bmatrix} \cos(\theta_4 - \frac{\pi}{2}) & 0 & \sin(\theta_4 - \frac{\pi}{2}) & 0 \\ \sin(\theta_4 - \frac{\pi}{2}) & 0 & -\cos(\theta_4 - \frac{\pi}{2}) & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.47}$$

For Link 5, the transformation matrix is given by:

$$A_5 = \begin{bmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & 0 \\ \sin(\theta_5) & \cos(\theta_5) & 0 & 0 \\ 0 & 0 & 1 & 190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.48}$$

The total transformation from the base to the end effector is obtained by multiplying the individual transformation matrices:

$$T = A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5 \tag{2.49}$$

2.3.2 Inverse Kinematics

The geometric representation of a robot arm involves describing the arm's structure in terms of its physical components and their spatial relationships. This includes the arm's links (rigid segments) and joints (points of rotation or translation). This representation is crucial for solving the inverse kinematics problem, which involves determining the necessary joint parameters to achieve a desired position and orientation of the robot's end effector (the tool or hand attached to the end of the arm) [1].
In order to effectively compute the inverse kinematics for our robotic arm, a clear geometric representation is essential. The Figure(2.10) illustrates the robotic arm's configuration, detailing the various joints and linkages that define its movement.

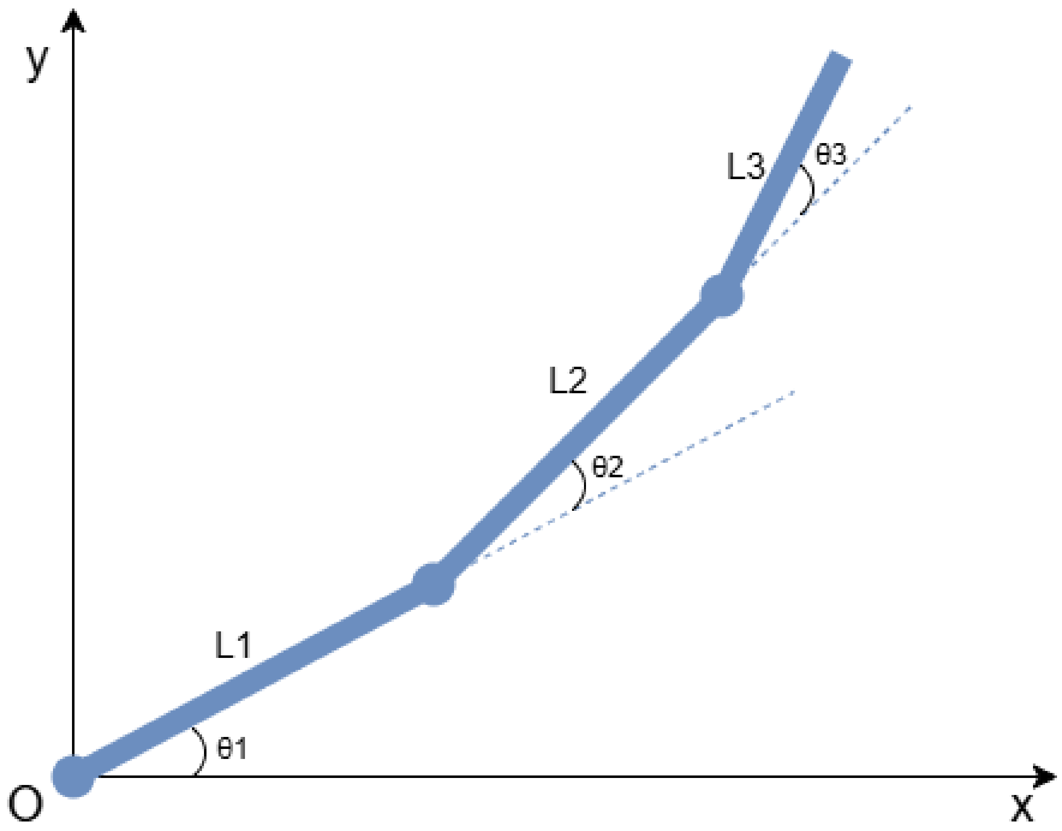


Figure 2.10: Geometric representation of the robot arm for inverse kinematics

To solve the inverse kinematics problem for a robotic arm with the given parameters $l_1 = 12.5$, $l_2 = 12.5$, and $l_3 = 6$, we start by expressing the end-effector coordinates (x_3, y_3) in terms of the joint angles θ_1, θ_2, and θ_3. The equations for these coordinates are given by:

$$\begin{cases} x_3 = l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2) + l_3\cos(\theta_1+\theta_2+\theta_3) \\ y_3 = l_1\sin(\theta_1) + l_2\sin(\theta_1+\theta_2) + l_3\sin(\theta_1+\theta_2+\theta_3) \end{cases} \tag{2.50}$$

and the orientation of the end-effector is defined as:

$$\varphi = \theta_1 + \theta_2 + \theta_3 \tag{2.51}$$

Given the values of x_3, y_3, and φ, the first step is to calculate the coordinates of the elbow joint (x_2, y_2). This can be done using the equations:

$$\begin{cases} x_2 = x_3 - l_3\cos(\varphi) \\ y_2 = y_3 - l_3\sin(\varphi) \end{cases} \tag{2.52}$$

Next, we calculate δ, which is the squared distance from the origin to the elbow joint:

$$\delta = x_2^2 + y_2^2 \tag{2.53}$$

To find θ_2, we use the cosine rule for triangles, which gives us:

$$\cos(\theta_2) = \frac{x_2^2 + y_2^2 - l_1^2 - l_2^2}{2l_1l_2} \tag{2.54}$$

The angle θ_2 can then be determined by:

$$\theta_2 = \arctan 2\left(\sqrt{1-\cos^2(\theta_2)}, \cos(\theta_2)\right) \tag{2.55}$$

We then move on to calculate θ_1 by using the coordinates of the elbow joint and the lengths of the links. From the equations of x_2 and y_2:

$$\begin{cases} x_2 = l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2) \\ y_2 = l_1\sin(\theta_1) + l_2\sin(\theta_1+\theta_2) \end{cases} \tag{2.56}$$

By substituting and rearranging, we obtain:

$$\begin{cases} x_2 = \cos(\theta_1)(l_1 + l_2\cos(\theta_2)) - \sin(\theta_1)(l_2\sin(\theta_2)) \\ y_2 = \sin(\theta_1)(l_1 + l_2\cos(\theta_2)) + \cos(\theta_1)(l_2\sin(\theta_2)) \end{cases} \tag{2.57}$$

These lead to the expressions for $\cos(\theta_1)$ and $\sin(\theta_1)$:

$$\cos(\theta_1) = \frac{(l_1 + l_2\cos(\theta_2))x_2 + l_2\sin(\theta_2)y_2}{x_2^2 + y_2^2} \tag{2.58}$$

$$\sin(\theta_1) = \frac{(l_1 + l_2\cos(\theta_2))y_2 - l_2\sin(\theta_2)x_2}{x_2^2 + y_2^2} \tag{2.59}$$

Finally, we calculate θ_1 using:

$$\theta_1 = \arctan 2\left(\sin(\theta_1), \cos(\theta_1)\right) \tag{2.60}$$

Once θ_1 and θ_2 are known, θ_3 can be determined straightforwardly from the equation:

$$\theta_3 = \varphi - \theta_1 - \theta_2 \tag{2.61}$$

This completes the inverse kinematics solution for the given robotic arm.

Chapter 3

Trajectory Generation and Planning

Trajectory planning is a crucial aspect of robotics, encompassing the process of determining a path or sequence of movements for a robot to follow in order to accomplish a specific task. This involves both mobile robots, which navigate through an environment, and robotic arms, which manipulate objects with precision. For mobile robots, trajectory planning ensures efficient and collision-free movement from one location to another, factoring in the robot's dynamics and environmental constraints. In the context of robotic arms, trajectory planning focuses on the smooth and accurate movement of the end-effector to perform tasks such as picking, placing, and assembling.
This chapter integrating advanced algorithms and control strategies, trajectory planning enables robots to operate

3.1 Cubic Polynomial Trajectory

Cubic polynomials are a method of moving the tool from its initial position to the desired position in a definite amount of time. It is a third-degree polynomial equation that is formulated as illustrated in Equation (3.1).

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{3.1}$$

The velocity and the acceleration profile along the trajectory of human gait are generated based on Equation (3.3) and Equation (3.3).

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2 \tag{3.2}$$

$$\ddot{\theta}(t) = 2a_2 + 6a_3 t \tag{3.3}$$

Furthermore, acceleration profile of cubic polynomials varies with time linearly but it is continuous. Thus, the trajectory does not require infinite accelerations.

The cubic polynomial trajectory can be represented as:

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

To find the coefficients a_0 through a_3, we impose the boundary conditions:

$$\begin{array}{cc} \theta(0) = \theta_{\mathrm{in}} & \theta(T) = \theta_{\mathrm{f}} \\ \dot{\theta}(0) = 0 & \dot{\theta}(T) = 0 \end{array}$$

To find the coefficients a_0, a_1, a_2, and a_3, we utilize the following constraints:

1. At $t = 0$:

$$\theta(0) = a_0 = \theta_{in}$$

$$\dot{\theta}(0) = a_1 = 0$$

2. At $t = T$:

$$\theta(T) = a_0 + a_1 T + a_2 T^2 + a_3 T^3 = \theta_f$$

$$\dot{\theta}(T) = a_1 + 2a_2 T + 3a_3 T^2 = 0$$

Solving for a_2 and a_3 yields:

$$a_3 = \frac{2(\theta_f - \theta_{in})}{T^3}$$

$$a_2 = -\frac{3a_3 T}{2}$$

Finally, substituting $a_0 = \theta_{in}$ and $a_1 = 0$, we obtain the complete trajectory equation. The complete trajectory equation is given by:

$$\theta(t) = \theta_{in} - \frac{3(\theta_f - \theta_{in})}{T^3} \cdot t^3 + 2 \cdot \frac{(\theta_f - \theta_{in})}{T^2} \cdot t^2$$

For example Figure 3.1 shows cubic trajectory with q0 = 10, qf = 20, t0=0, tf =1 and V0 = 0, Vf =0. The corresponding angle, velocity and acceleration curves are given in Figures 3.1.

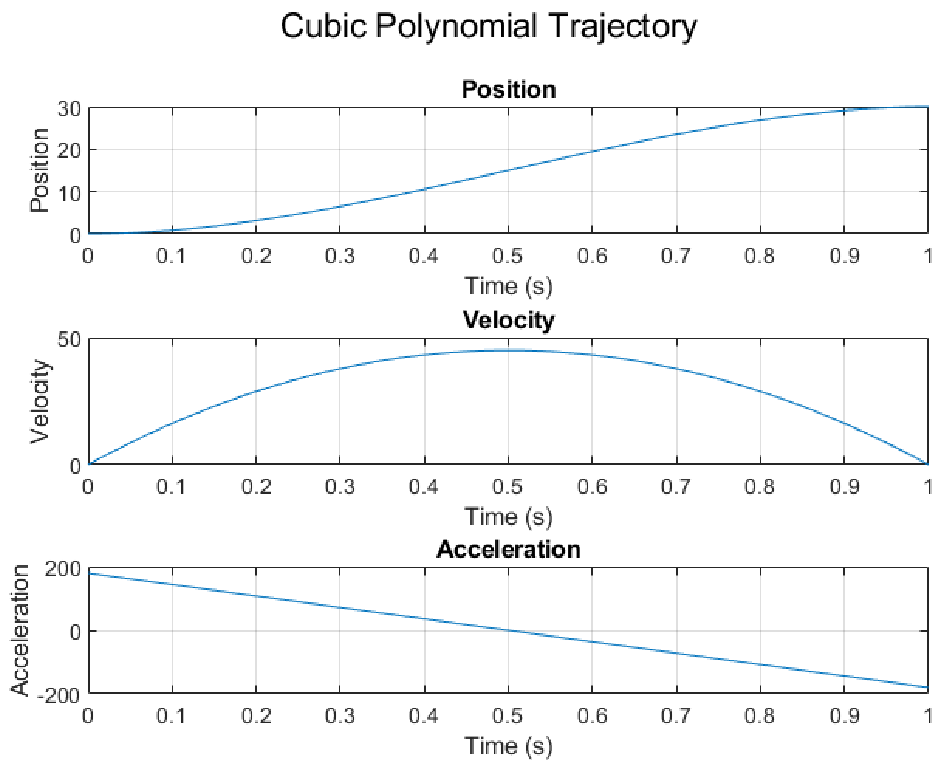


Figure 3.1: Cubic Polynomial Trajectory

3.2 Quintic Polynomial Trajectory

A cubic trajectory produces discontinuities in the acceleration but continuous positions and velocities at the start and finish points, as shown in Figure 3.1. The term "jerk" refers to the derivative of acceleration. An abrupt jolt caused by a discontinuity in acceleration might activate vibration modes in the manipulator and lower tracking accuracy.
To ensure precise control, it may be necessary to define limitations not only on the position and velocity but also on the acceleration. In such instances, a total of six constraints are imposed, encompassing the initial and final configurations, velocities, and accelerations. Consequently, a fifth order polynomial is essential.
The quintic polynomial equation is a fifth-degree polynomial equation which is formulated as shown in Equation (3.4). The quintic polynomial equation can be used to represent angular position profile of hip, knee and ankle joints.

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{3.4}$$

By differentiating Equation (3.4), the velocity profile is obtained as shown in Equation (3.5).

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \tag{3.5}$$

By differentiating Equation (3.5), the acceleration profile is obtained based on Equation (3.6).

$$\ddot{\theta}(t) = 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3 \tag{3.6}$$

To plan a quintic polynomial trajectory satisfying the given boundary conditions, we can define the trajectory as shown in Equation (3.4).

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

To find the coefficients a_0 through a_5, we impose the boundary conditions:
Quintic polynomial trajectory planning involves designing a path such that the initial and final positions are set to specific values, while ensuring that both the velocities and accelerations at these points start and end at zero.

$$\begin{aligned} \theta(0) &= \theta_{\text{in}} & \qquad \theta(T) &= \theta_{\text{f}} \\ \dot{\theta}(0) &= 0 & \qquad \dot{\theta}(T) &= 0 \\ \ddot{\theta}(0) &= 0 & \qquad \ddot{\theta}(T) &= 0 \end{aligned}$$

Let's solve these step by step:

$$\begin{aligned} a_0 &= \theta_{\text{in}} \\ a_1 &= 0 \\ a_2 &= 0 \\ a_3 &= \frac{10(\theta_{\text{f}} - \theta_{\text{in}})}{T^3} \\ a_4 &= -\frac{15(\theta_{\text{f}} - \theta_{\text{in}})}{T^4} \\ a_5 &= \frac{6(\theta_{\text{f}} - \theta_{\text{in}})}{T^5} \end{aligned}$$

Therefore, the quintic polynomial trajectory satisfying the given boundary conditions is:

$$\theta(t) = \theta_{\text{in}} + \frac{10(\theta_{\text{f}} - \theta_{\text{in}})}{T^3} t^3 - \frac{15(\theta_{\text{f}} - \theta_{\text{in}})}{T^4} t^4 + \frac{6(\theta_{\text{f}} - \theta_{\text{in}})}{T^5} t^5$$

This trajectory ensures that the initial and final positions, velocities, and accelerations are all zero.
Figure 3.2 shows a quintic polynomial trajectory with q(0) = 0, q(2) = 40 with zero initial and final velocities and accelerations.

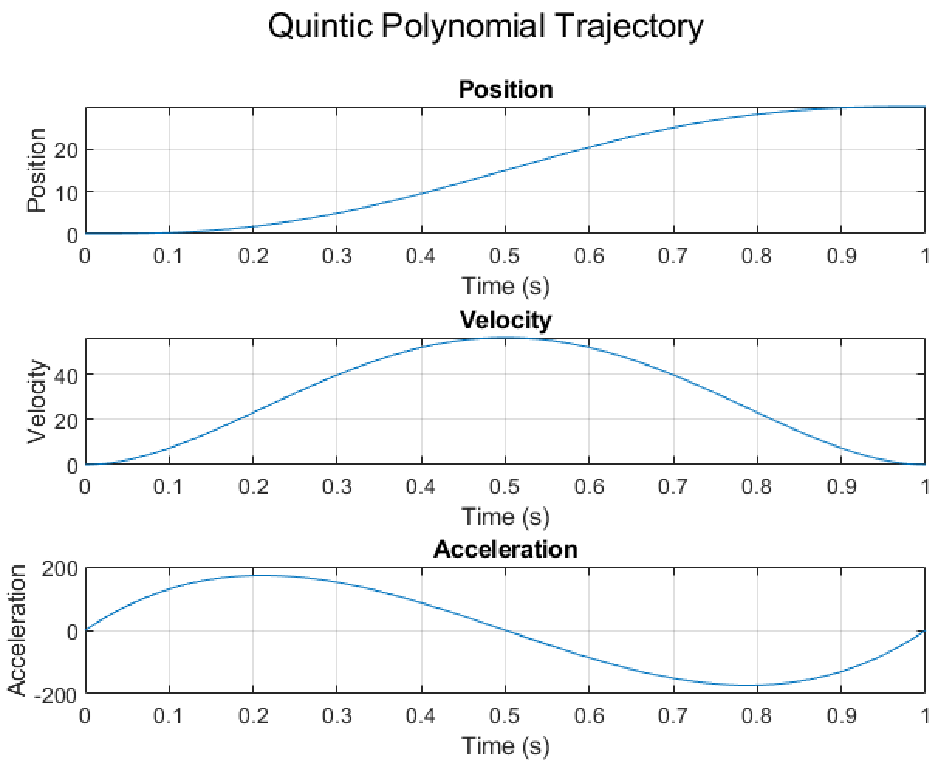


Figure 3.2: Quintic Polynomial Trajectory

Comparison of cubic and quintic polynomial trajectories

The following table 3.1 offers a concise comparison between cubic and quintic polynomial trajectories, two common mathematical models used in various fields. By examining their differences in degree, smoothness, boundary conditions, computational complexity, and accuracy, we can better understand their distinct characteristics and applications. This comparison serves as a helpful reference for researchers, engineers, and mathematicians seeking to choose the most appropriate polynomial trajectory for their specific needs and constraints.

Aspect	**Cubic Polynomial Trajectory**	**Quintic Polynomial Trajectory**
Degree of Polynomial	Third-degree ($ax^3 + bx^2 + cx + d$)	Fifth-degree ($ax^5 + bx^4 + cx^3 + dx^2 + ex + f$)
Smoothness	Less smooth	Smoother
Boundary Conditions	Fewer boundary conditions required	More boundary conditions required
Computational Complexity	Lower	Higher
Accuracy	Less accurate	More accurate

Table 3.1: Comparison of cubic and quintic polynomial trajectories

3.3 Dijkstra's Algorithm and A* Algorithm

In the field of robotics, motion planning is a critical challenge, particularly in navigating robots through complex environments. Two of the most prominent algorithms used for pathfinding in robot motion planning are Dijkstra's Algorithm and the A* Algorithm. These algorithms form the backbone of many navigation systems, enabling robots to determine the most efficient path from a start point to a goal point, while avoiding obstacles and optimizing various criteria, such as distance, time, or energy consumption. Understanding these algorithms is essential for developing advanced robotic systems capable of autonomous navigation.

3.3.1 History

The development of pathfinding algorithms has significantly influenced the field of robotics and artificial intelligence. This section provides a historical perspective on both Dijkstra's Algorithm and the A* Algorithm, tracing their origins, key developments, and impact on technology.

Dijkstra's Algorithm

Dijkstra's Algorithm, named after the Dutch computer scientist Edsger W. Dijkstra, was first introduced in 1956 and later published in 1959. The algorithm was initially conceived as a method for finding the shortest path in a graph, a common problem in various fields such as transportation, telecommunication, and network routing [11]. Dijkstras inspiration for the algorithm stemmed from a need to find the shortest route in a city, and his solution has since become one of the most widely used algorithms in computer science. Over the decades, Dijkstras Algorithm has found applications in numerous domains, including robotics, where it is utilized for motion planning and navigation in grid-based environments. A picture of Edsger W. Dijkstra is shown in Figure (3.3).

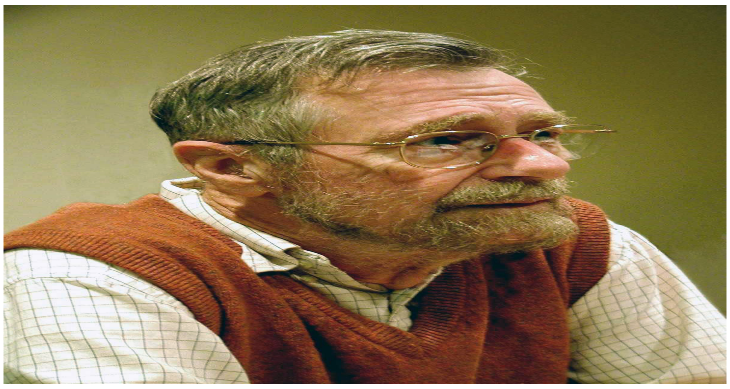


Figure 3.3: Edsger Wybe Dijkstra

A* Algorithm

The A* Algorithm, developed by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968, extended the principles of Dijkstra's Algorithm by incorporating heuristic techniques to improve search efficiency [12]. A* gained prominence through its application in Shakey the Robot, an early autonomous robot developed at the Stanford Research Institute. Shakey's successful navigation capabilities showcased A*'s ability to handle complex environments and dynamic scenarios effectively. The implementation of A* in Shakey demonstrated the algorithms practical value in real-world robotics [13]. A notable image of Shakey the Robot is shown in Figure (3.4).

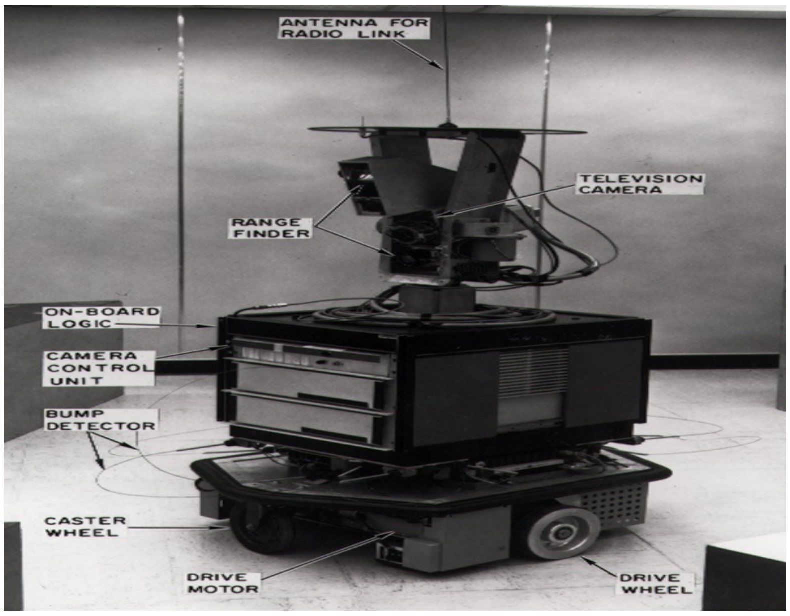


Figure 3.4: Shakey the Robot

3.3.2 Description

Understanding how Dijkstra's and A* algorithms function is crucial for their effective implementation in robotics. This section describes the operational principles of both algorithms, highlighting their methodologies and distinguishing features.

Dijkstra's Algorithm

Dijkstras Algorithm is a graph-based approach that solves the single-source shortest path problem for a graph with non-negative edge weights. The algorithm works by systematically exploring all possible paths from the start node to all other nodes, selecting the path with the smallest cumulative distance at each step. It maintains a priority queue to keep track of the shortest known distances to each node and updates these distances as it discovers shorter paths. The algorithm guarantees finding the shortest path to the goal node by the time it reaches it. However, this exhaustive search process can be computationally expensive, especially in large graphs, making it less efficient in certain applications like real-time robotics [14].

A* Algorithm

The A* Algorithm builds upon the principles of Dijkstras Algorithm by introducing a heuristic function that estimates the cost from a given node to the goal. This heuristic guides the search process, allowing the algorithm to prioritize exploring paths that appear to lead closer to the goal. A* maintains two main functions: the cost from the start node to the current node (known as the "g" value) and the estimated cost from the current node to the goal (known as the "h" value). The sum of these two values, known as the "f" value, is used to determine the order in which nodes are explored. By balancing exploration and goal-directed search, A* often finds the shortest path more quickly than Dijkstras Algorithm, particularly in environments where the heuristic is well-chosen [15].

3.3.3 Pseudocode

In this section, we present the pseudocode for Dijkstra's Algorithm and the A* Algorithm. Pseudocode serves as a high-level representation of the algorithmic steps, providing a clear and concise way to understand the core logic and operations without delving into specific programming syntax. For each algorithm, the pseudocode outlines the essential steps and decision-making processes involved in finding the shortest path in a graph. This approach helps in grasping the underlying principles and mechanisms before implementing the algorithms in actual code.

Dijkstra's Algorithm

Dijkstra's Algorithm is a fundamental technique used to find the shortest path from a starting node to all other nodes in a weighted graph. The algorithm initializes distances to all nodes as infinity, except for the starting node, which is set to zero. Using a priority queue to explore nodes with the smallest known distance first, it iteratively updates the shortest known distances to each node based on the weights of the edges. This process continues until all reachable nodes have their shortest distances determined. The result is a mapping of each node to its minimum distance from the start node.
The pseudocode for Dijkstras Algorithm is as follows:

Algorithm 1 Dijkstra's Algorithm

```
function Dijkstra(Graph, start)
    dist ← map with default value Infinity
    dist[start] ← 0
    priority_queue ← priority queue containing start with priority 0
while priority_queue is not empty do
   u ← node in priority_queue with smallest distance
   remove u from priority_queue
   for each neighbor v of u do
      alt ← dist[u] + weight(u, v)
      if alt < dist[v] then
         dist[v] ← alt
         add v to priority_queue with priority alt
      end if
   end for
end while
return  dist =0
```

A* Algorithm

The A* Algorithm is an efficient pathfinding technique that combines elements of Dijkstra's Algorithm with heuristic-based approaches to find the shortest path from a starting node to a goal node. It maintains a priority queue of nodes to explore, prioritizing those with the lowest estimated total cost, which is the sum of the cost from the start node to the current node and a heuristic estimate of the cost to the goal. During each iteration, the algorithm evaluates neighboring nodes, updating their scores and path information if a shorter path is found. The process continues until the goal node is reached, at which point the optimal path is reconstructed. If the open set is exhausted without finding the goal, the algorithm reports failure.
The pseudocode for the A* Algorithm is as follows:

Algorithm 2 A* Algorithm

```
function A*(Graph, start, goal)
    open_set ← priority queue containing start with priority 0
    came_from ← map with default value undefined
    g_score ← map with default value Infinity
    g_score[start] ← 0
    f_score ← map with default value Infinity
    f_score[start] ← heuristic(start, goal)
while open_set is not empty do
    current ← node in open_set with lowest f_score
    if current == goal then
        return reconstruct_path(came_from, goal)
    end if
    remove current from open_set
    for each neighbor in neighbors(current) do
        tentative_g_score ← g_score[current] + distance(current, neighbor)
        if tentative_g_score < g_score[neighbor] then
            came_from[neighbor] ← current
            g_score[neighbor] ← tentative_g_score
            f_score[neighbor] ← g_score[neighbor] + heuristic(neighbor, goal)
            if neighbor not in open_set then
                add neighbor to open_set with priority f_score[neighbor]
            end if
        end if
    end for
end while
return failure =0
```

3.3.4 Comparison

While both Dijkstras Algorithm and A* Algorithm are widely used for pathfinding in robotics, they differ significantly in their approach and efficiency. Dijkstras Algorithm guarantees the shortest path by exploring all possible routes, which can be computationally expensive, especially in large environments. It is particularly well-suited for scenarios where all paths need to be explored, such as in network routing. However, its lack of heuristic guidance makes it less efficient for real-time applications in robotics [16].
The A* Algorithm, on the other hand, incorporates heuristic estimates that guide the search process, making it faster in many cases. This makes A* more suitable for dynamic environments where quick decision-making is crucial. The efficiency of A* is heavily dependent on the quality of the heuristic function; a well-designed heuristic can significantly reduce the search space, while a poorly chosen one can lead to suboptimal performance.

Chapter 4

Simulation

MATLAB is an essential tool in the field of robotics, offering a comprehensive environment for the simulation, modeling, and control of robotic systems. It provides a range of toolboxes and libraries, such as the Robotics System Toolbox and Simulink, which facilitate the development and testing of complex robotic algorithms. MATLAB supports various aspects of robotics, including kinematic and dynamic modeling, path planning, control system design, and sensor integration. Its capabilities extend to 3D visualization, enabling detailed simulations of robotic movements and interactions within different environments. Furthermore, MATLAB's machine learning and AI toolboxes empower the development of intelligent robotic systems capable of tasks like object recognition and autonomous decision-making. Practical applications of MATLAB in robotics include simulations for mobile robots, robotic arms, UAVs, and AUVs, showcasing its versatility. The integration of high-level programming, extensive documentation, and advanced visualization tools makes MATLAB a powerful and user-friendly platform for pushing the boundaries of robotics research and development.

4.1 Simulation of Robotic Manipulator Arm Using MATLAB

The simulation of robotic manipulator arms using MATLAB is essential for analyzing and optimizing robotic systems prior to physical testing. MATLAB provides a robust platform for modeling, simulating, and analyzing robotic manipulators through tools like Simulink and Simscape Multibody. This approach allows engineers to create accurate digital models of robotic arms, evaluate their performance, and refine control algorithms within a virtual environment. The use of MATLAB in simulation enhances design efficiency, reduces prototyping costs, and supports the development of effective control strategies, making it a crucial tool in modern robotics engineering.

The provided MATLAB script (4.1) is utilized for modeling and simulating a robotic manipulator using the Robotics Toolbox, with the Denavit-Hartenberg (DH) convention

employed for kinematic modeling. Initially, the script defines the physical dimensions of the robot's links through the parameters L_1, L_2, L_3, and L_4.
The code constructs a series of 'Link' objects, each representing a segment of the robotic arm. These objects are parameterized by joint angles (θ), link offsets (d), link lengths (a), and link twists (α) according to the DH parameters. For instance, the first link is specified with a zero joint angle, a link offset of L_1, and a twist of $-\pi/2$. The subsequent links are parameterized similarly, reflecting the robot's kinematic structure and joint configurations.
A 'SerialLink' object, named 'Braccio', is created to represent the complete robotic arm as a series of these defined links. The 'teach()' method is invoked on this object, launching an interactive graphical user interface Figure (4.1). This GUI facilitates the visualization and manipulation of the robot's joint angles, thereby aiding in the analysis and study of the robot's kinematic behavior. This script serves as a foundational tool for exploring robotic arm kinematics in both educational and research contexts [17].

```matlab
clear all; close all; clc;

% Define the lengths of the robot arm segments
L1=70;
L2=125;
L3=125;
L4=190;

% Define the robot links using the Link class
% L = Link([theta d a alpha]) - Denavit-Hartenberg parameters
L(1) = Link ([0 L1 0 -pi/2])
L(2) = Link ([0+(pi/2) 0 -L2 0])
L(3) = Link ([0 0 -L3 0])
L(4) = Link ([0-(pi/2) 0 0 pi/2])
L(5) = Link ([0 L4 0 0])

Rob=SerialLink(L); % Create the SerialLink object
Rob.name = 'Robot_Manipulator' % Set the name of the robot
Rob.teach() % Visualize and interact with the robot
```

Listing 4.1: Defining and Visualizing a Robotic Arm in MATLAB

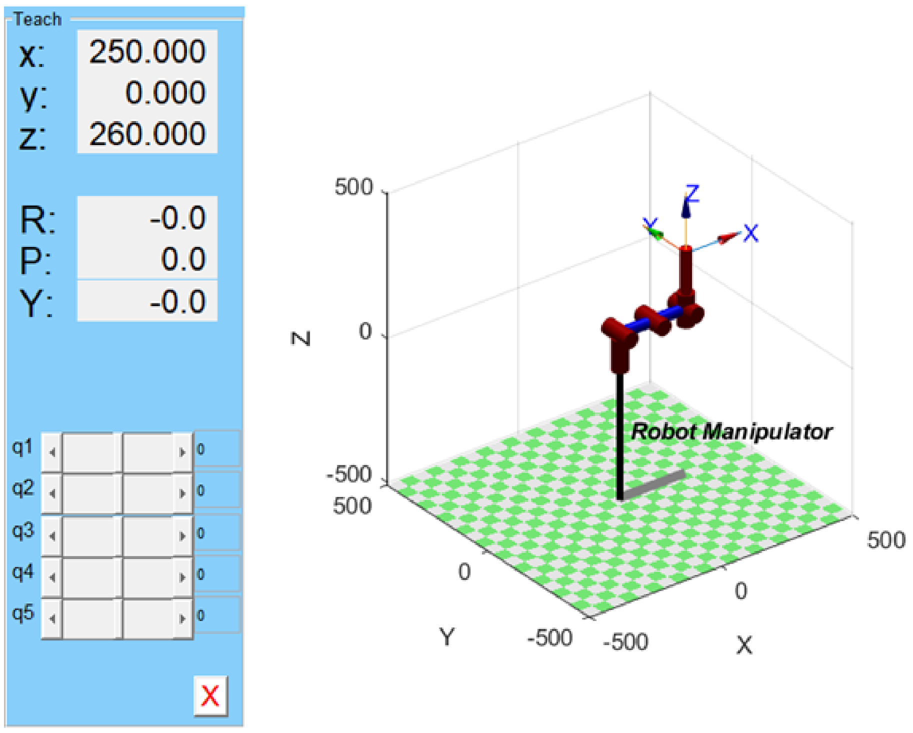


Figure 4.1: Manipulating Robot Arm with the teach() GUI

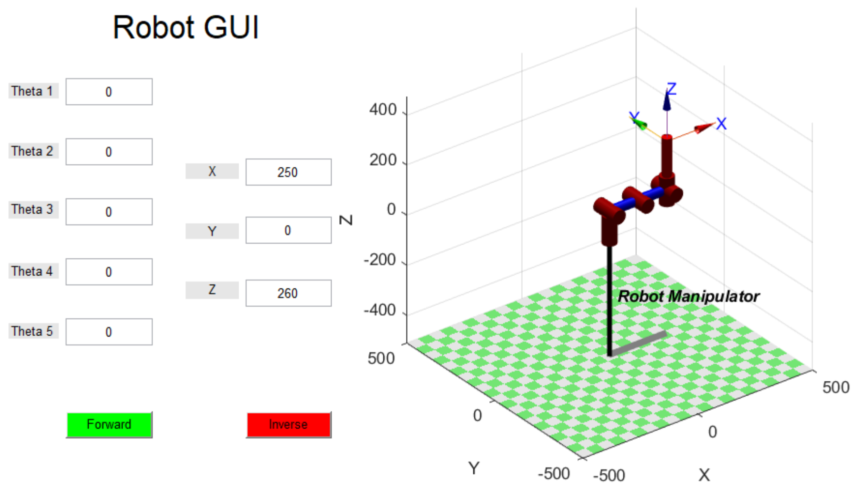
The MATLAB GUI for robotic manipulators offers a comprehensive tool for performing and visualizing kinematic calculations. This interface, created with MATLAB's GUIDE, integrates buttons, text fields, and graphical plots to enable users to input joint angles and target positions, observing the robot's movements in real-time.

In forward kinematics, the GUI calculates the position of the robot's end-effector based on user-defined joint angles. It converts these angles from degrees to radians and utilizes the Denavit-Hartenberg parameters to model the robot's links. The trajectory of the end-effector is computed using a quintic polynomial function, which smooths the motion by controlling acceleration and deceleration.

The equation, given by $\boldsymbol{q(t)=10\left(\frac{t}{T_f}\right)^3-15\left(\frac{t}{T_f}\right)^4+6\left(\frac{t}{T_f}\right)^5}$, ensures continuous and smooth movement. These calculations are implemented in the GUI using the `Robot.fkine` method, a MATLAB function that computes the forward kinematics by generating the homogeneous transformation matrix for the end-effector. This matrix encodes the position and orientation of the end-effector relative to the robot's base frame, which is then visualized in a 3D plot, providing a dynamic representation of the robot's trajectory.

Inverse kinematics, on the other hand, determines the necessary joint angles to achieve a specified end-effector position. The GUI retrieves the target position, and a similar quintic polynomial trajectory is used to interpolate between an initial and target position. Intermediate positions are calculated using linear interpolation, and the corresponding joint angles are computed using the `Robot.ikine` method. This function iteratively solves for the joint angles that minimize the error between the desired and actual positions of the end-effector, considering the robot's constraints. These joint angles are then applied to the robot model in the GUI, updating the visualization as the robot moves toward the target position.

As shown in Figure (4.2), the MATLAB GUI provides an interactive platform for robotic arm control and trajectory planning.

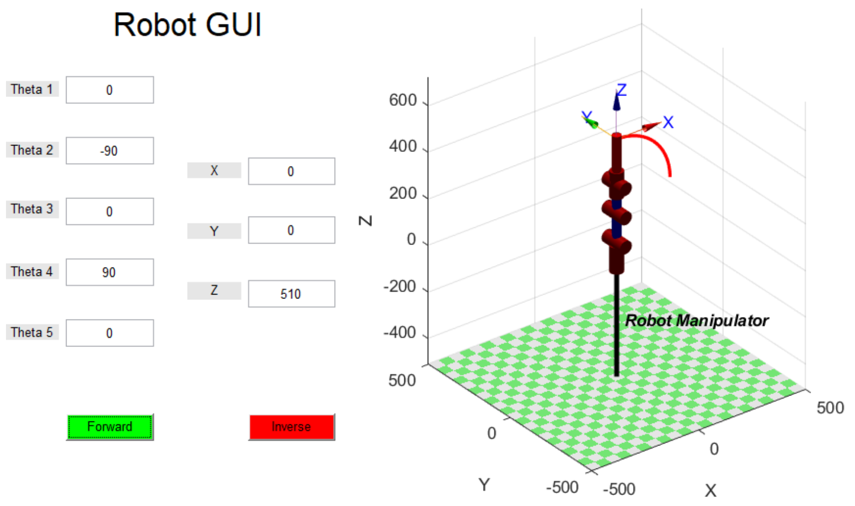

Robot GUI

Theta 1	0
Theta 2	0
Theta 3	0
Theta 4	0
Theta 5	0
X	250
Y	0
Z	260

Forward
Inverse

Figure 4.2: MATLAB GUI for Robotic Arm Kinematics and Quintic Polynomial Trajectory Planning

The MATLAB GUI for the robotic arm kinematics, which illustrates the implementation of a Quintic Polynomial Trajectory for the joint angles (0ř -90ř 0ř 90ř 0ř), is depicted in Figure (4.3).


Robot GUI

Theta 1	0
Theta 2	-90
Theta 3	0
Theta 4	90
Theta 5	0
X	0
Y	0
Z	510

Forward
Inverse

Figure 4.3: MATLAB GUI for Quintic Polynomial Trajectory Execution in Robotic Arm with Joint Angle Sequence (0ř -90ř 0ř 90ř 0ř)

Modeling and simulating a robotic arm using MATLAB Simscape Multibody and SolidWorks involves a systematic approach that combines mechanical design with dynamic analysis [18]. The process begins with creating a detailed 3D model of the robotic arm in SolidWorks, including all necessary components such as links, joints, and actuators, with precise dimensions and constraints. Once the design is complete, the SolidWorks model is exported as an XML or STEP file. In MATLAB, this file is imported using the **`smimport(filename.xml)`**command in the MATLAB command window, which automatically generates the associated Simulink program for the assembly. This facilitates a seamless transition from SolidWorks to MATLAB.

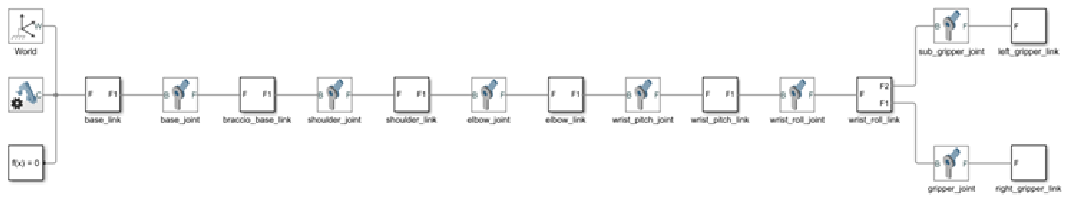


Figure 4.4: Robot Manipulator Multi Body

The Simulink model is constructed using Simscape Multibody blocks that replicate the mechanical structure of the robotic arm. Dynamics such as masses, inertias, and joint limits are defined to simulate physical interactions. Forward and inverse kinematics algorithms are implemented in MATLAB code and integrated into the Simulink model to control the arms movements, with PID controllers or other strategies employed to ensure accurate trajectory following.
The simulation is then executed, and results are visualized using Simulink scopes and MATLAB graphs, allowing for the comparison of actual performance with the desired trajectory. This approach enables thorough analysis and adjustment of the robotic arms performance, ensuring precise simulation and control.

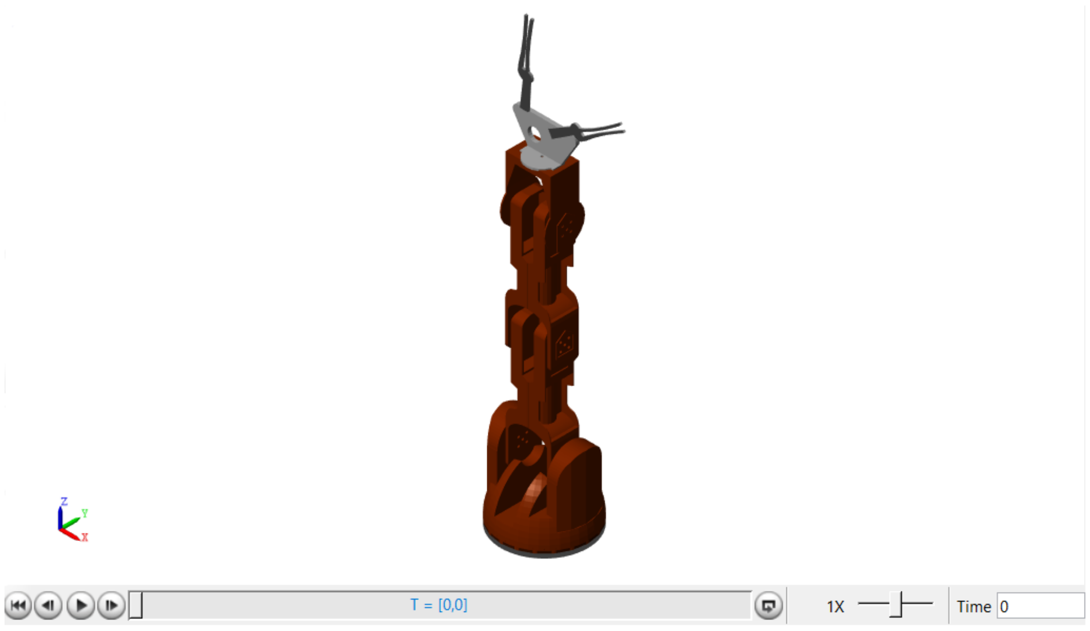


Figure 4.5: Robot Manipulator Arm 3D Multi Body

To illustrate the application of quintic polynomial trajectories in robotic manipulator simulations, consider moving the arm from an initial position $(0°, 0°, 0°, 0°, 0°, 0°)$ to a final position $(-90°, 45°, 45°, 45°, 0°, 45°)$. The trajectory is defined by a quintic polynomial, which ensures smooth and continuous motion between these positions. This polynomial trajectory specifies initial and final positions, velocities, and accelerations, allowing for a precise and smooth transition.In the Simulink model, this trajectory is implemented to compute the required joint angles over time. The resulting motion is then analyzed to ensure that the manipulator follows the desired path accurately and smoothly.

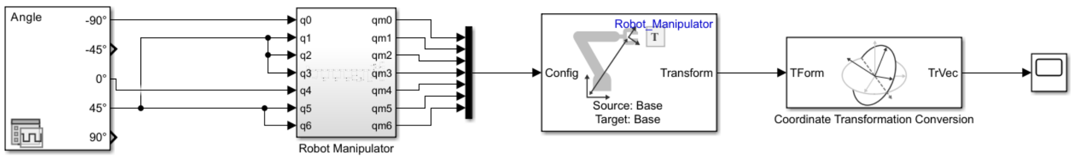


Figure 4.6: Robot Manipulator Arm Multi Body With Quintic Polynomial Trajectories

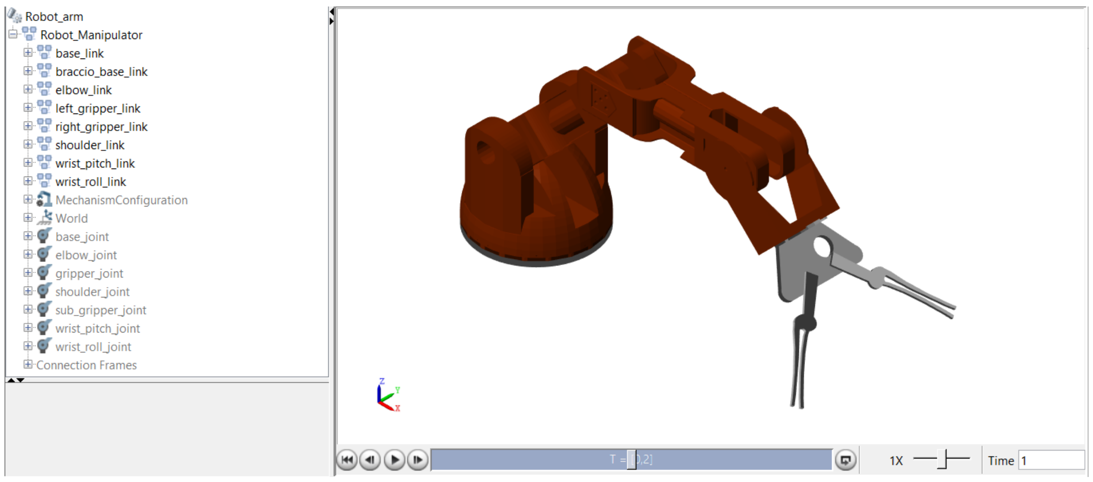


Figure 4.7: Robot Manipulator Arm 3D Multi Body With Quintic Polynomial Trajectories

In robotics, inverse kinematics is used to compute the joint angles required for a robot to achieve a target position and orientation of its end-effector. In Simulink and MATLAB's Simscape Multibody, this process is modeled through a combination of tools. The Signal Editor defines the desired motion, while Coordinate Transformation blocks handle frame conversions. The Inverse Kinematics block calculates the necessary joint movements, and the Robot Manipulator block simulates these actions in real-time. Additional blocks, such as *Get Transform* and *Coordinate Transformation Conversion*, ensure precise visualization of the robot's motion in various coordinate frames. This approach allows for accurate simulation and control of robotic systems in dynamic environments, as illustrated in Figure (4.8).

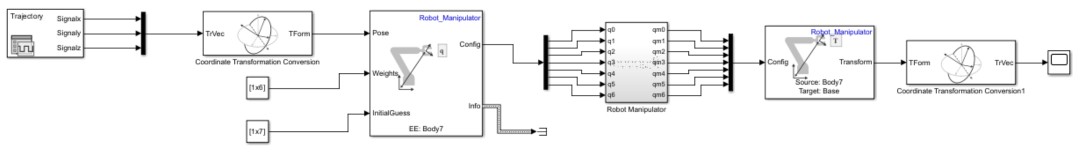


Figure 4.8: Inverse Kinematics Control of a Robot Arm in Simulink using Simscape Multibody

This example highlights the effectiveness of inverse kinematics for achieving complex manipulative tasks and validating the robotic arms performance in a simulated setting, as shown in Figure (4.9).

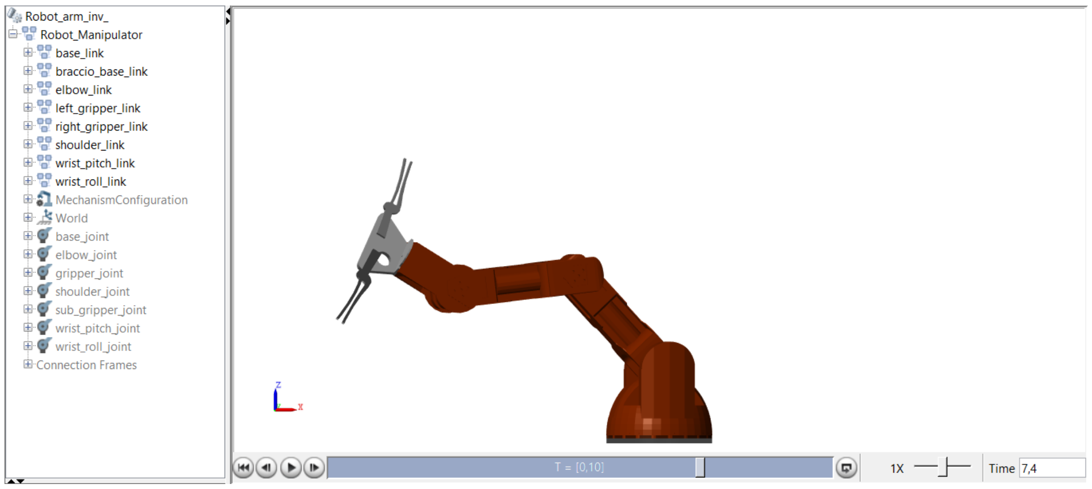


Figure 4.9: Inverse Kinematics Control of a Robot Arm in Simulink using Simscape Multibody

4.2 Simulation of Robotic Manipulator Arm Using GAZEBO

Robot Operating System (ROS) has emerged as a pivotal framework in the field of robotics, providing a flexible and modular platform for developing complex robotic systems. ROS is not an operating system in the traditional sense but rather a middleware that facilitates communication between various components of a robotic system, such as sensors, actuators, and control algorithms. This modularity allows for the reuse of code across different projects and supports the integration of diverse hardware and software components. With its extensive library of tools and packages, ROS has become the de facto standard for both research and industrial applications, enabling the rapid prototyping and deployment of sophisticated robotic systems.

ROS

Robot Operating System

Figure 4.10: ROS : Emblem of Robotics Operating System

MoveIt is a versatile framework in the ROS (Robot Operating System) ecosystem, designed to simplify motion planning, manipulation, and perception for robotic systems. It supports robots with multiple degrees of freedom, such as robotic arms and mobile manipulators. MoveIt offers advanced motion planning algorithms for generating collision-free trajectories, integrated with real-time perception to adapt to dynamic environments. Its modular architecture allows for customization, and it includes tools for inverse and forward kinematics. Additionally, MoveIt supports simulation environments for testing robotic motions before real-world deployment, making it valuable in both research and industry.

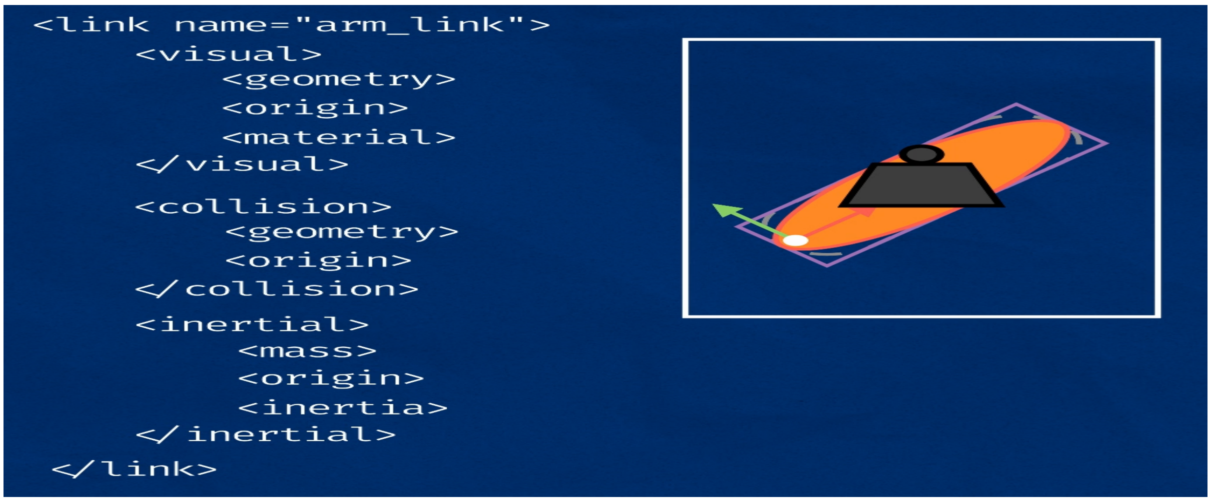
MoveIt

Figure 4.11: MoveIt : Symbol of Robotics Motion Plannin

Complementing ROS is the Gazebo 3D simulator, a powerful tool for simulating and testing robotic systems in a virtual environment. Gazebo offers a high-fidelity simulation environment that replicates the physical properties of the real world, including dynamics, kinematics, and sensor feedback. This allows researchers and developers to test and validate their algorithms and designs before deploying them on physical robots, thus reducing the risk of hardware damage and optimizing the development process. The integration of Gazebo with ROS provides a seamless workflow, where the same ROS code can be used both in simulation and on real hardware, facilitating a smooth transition from development to deployment.

Figure 4.12: Gazebo : Symbol of Robotics Simulation Environment

To model and analyze a robotic arm within the Gazebo simulation environment, one must first develop a detailed 3D representation of the robotic arm using CAD software such as SolidWorks or Blender. This model should encompass all critical components, including links, joints, and actuators, each accurately defined with precise dimensions, mass properties, and motion constraints. Following the completion of the model, it is exported in a format compatible with Gazebo, typically as a URDF (Unified Robot Description Format) file.
Subsequently, the URDF model is imported into Gazebo, where it is configured for simulation. This involves setting up the environment to accurately reflect the robotic arm's dynamics by adjusting physics parameters such as gravity, friction, and collision properties. Gazebos advanced physics engine facilitates a realistic representation of these dynamics, enabling detailed evaluation of the robotic arms behavior under various conditions [19].
The general layout of a link tag is shown in Figure (4.13)



```
<link name="arm_link">
    <visual>
        <geometry>
        <origin>
        <material>
    </visual>
    <collision>
        <geometry>
        <origin>
    </collision>
    <inertial>
        <mass>
        <origin>
        <inertia>
    </inertial>
</link>
```

Figure 4.13: Overall Structure of a Link Tag

Integration with ROS (Robot Operating System) further enhances the simulation process. ROS provides robust tools for controlling the robotic arm, visualizing the simulation, and collecting performance data. This integration supports the testing of complex control strategies, including inverse kinematics and path planning algorithms, within the

simulated environment. Such comprehensive simulation enables researchers to refine and validate the robotic arms design and functionality, mitigating potential issues before physical implementation and ensuring optimal performance in practical applications. Once the URDF file of the robotic arm is loaded into Gazebo, the next step is to launch and visualize the model in the simulation environment. This involves initializing the Gazebo world and launching it using a ROS launch file, which includes both the Gazebo world and the URDF model. The launch file defines key parameters such as the robot's initial position and environment settings.
Once Gazebo is running, the robotic arm will be visible, and its components can be visualized and controlled. ROS nodes can be launched to manage joint movements, sensor feedback, and task execution, enabling real-time interaction with the arm. Gazebo also offers tools for monitoring joint states and forces, making it ideal for testing robotic systems.

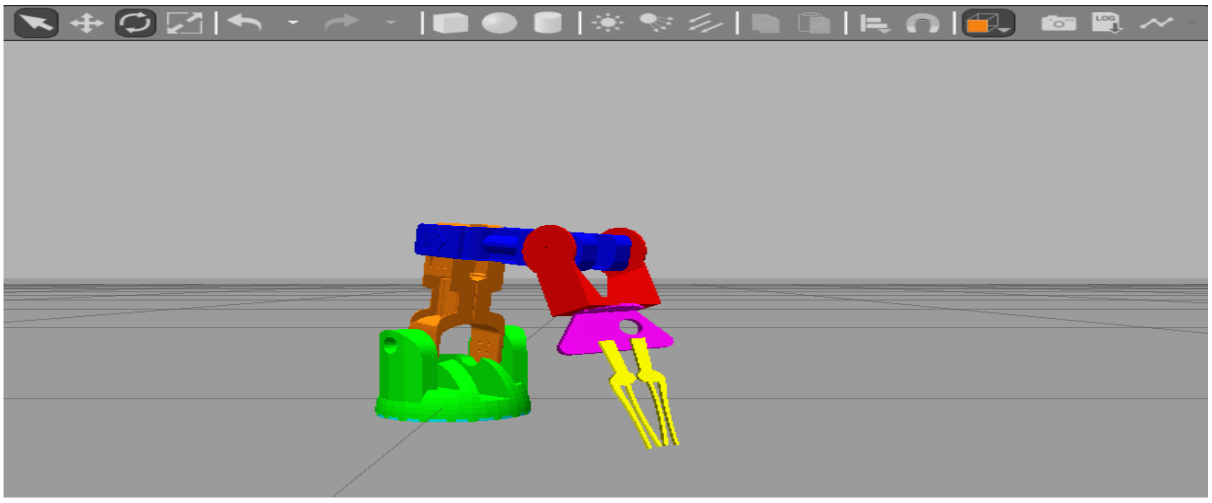


Figure 4.14: Robotic Arm Deployment and Visualization in Gazebo ROS

To simulate pick-and-place operations using ROS (Robot Operating System), MoveIt, and Gazebo, several key components are integrated to enable autonomous object manipulation. The pick operation involves identifying and grasping an object, such as a red block, using a robotic arm in a simulated environment.

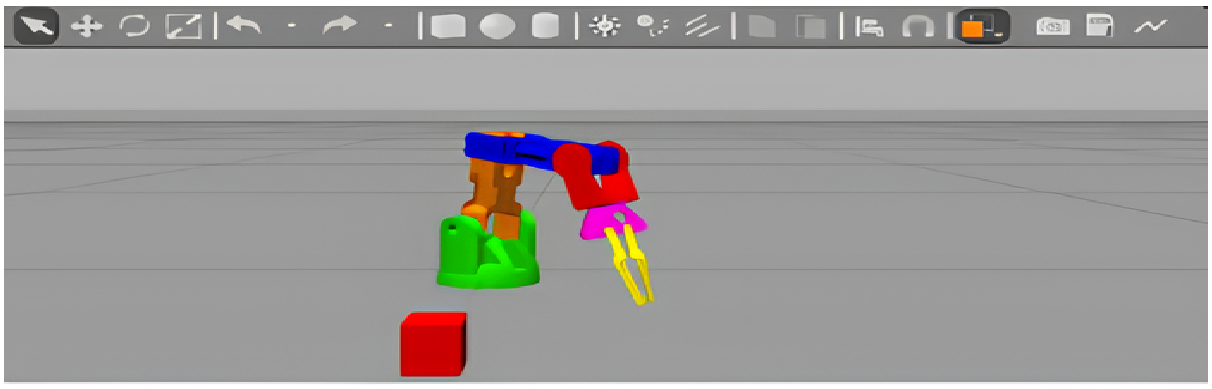


Figure 4.15: Visualization of Robotic Arm Interacting with Red Block in Gazebo ROS

Utilizing ROS and MoveIt, the robot arm is capable of identifying and picking up the red block from various positions. The arm can approach the block either from above or from the side, depending on its location. The process begins with the ROS MoveIt framework, which plans the motion trajectory for the arm. The trajectory planning involves calculating the optimal path to reach the block while avoiding any obstacles. Once the path is determined, the Gazebo simulation environment executes the planned motion, allowing the Braccio arm to successfully grasp the red block. This simulation demonstrates the effectiveness of ROS and MoveIt in performing precise and adaptable pick operations, showcasing the potential for advanced robotic.

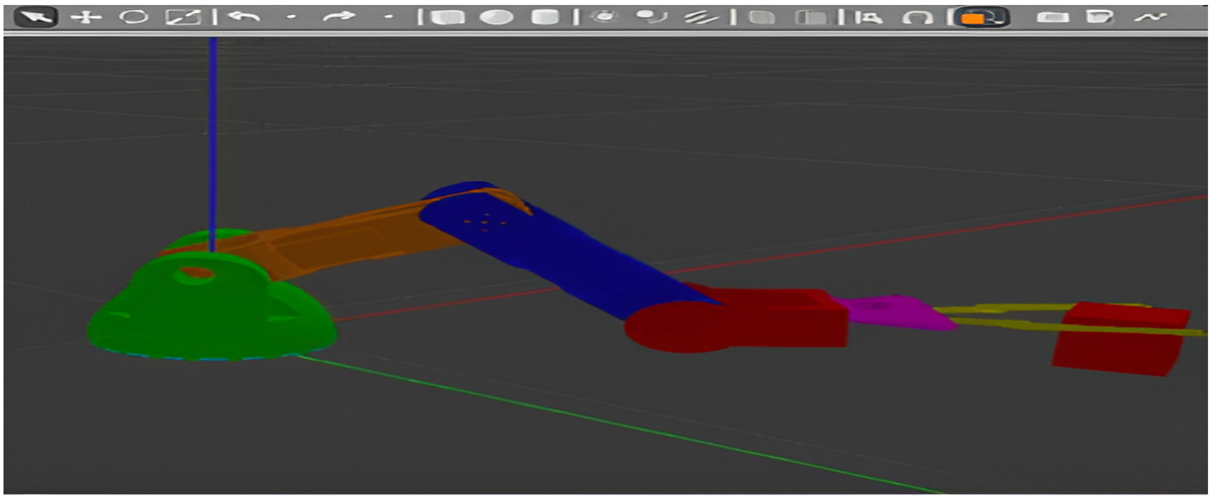


Figure 4.16: Execution of Red Block Pickup by Robot Arm with ROS and MoveIt

4.3 Simulation of Robotic Mobile Using MATLAB

In this section, we delve into the implementation of forward and inverse kinematics for a mobile robot equipped with Mecanum wheels. The robot's movement capabilities are modeled using parameters such as wheel radius, wheelbase (the distance between the front and rear wheels), and track width (the distance between the left and right wheels). We present two approaches: forward kinematics and inverse kinematics.
The robot's initial position is defined by the coordinates $[x_0, y_0]$ and the orientation θ_0, where x_0 and y_0 specify its location on the Cartesian plane, and θ_0 represents its angular orientation relative to a fixed reference axis. These parameters serve as the starting point for analyzing the robot's movement in subsequent time steps. Figure (4.17) illustrates the robot's initial configuration, where the position is represented as $[x, y, \theta] = [0, 0, 0]$, indicating that the robot starts at the origin with zero orientation.

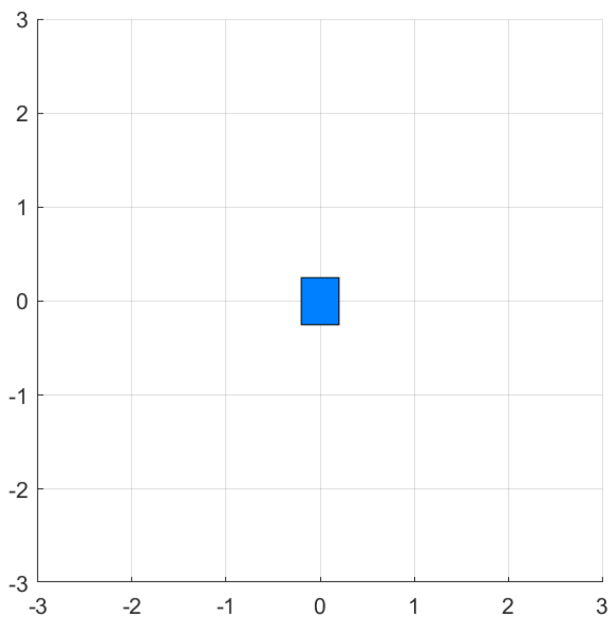
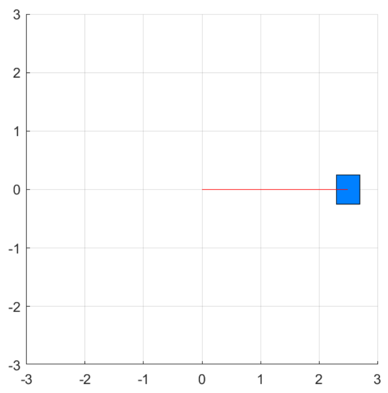
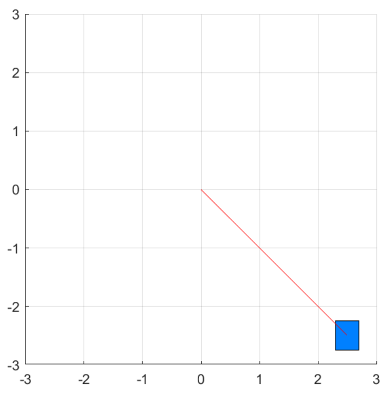


Figure 4.17: Initial position and orientation of the robot at $[x_0, y_0, \theta_0]$.

The first simulation demonstrates forward kinematics for a four-wheel Mecanum robot. This method computes the robots motion based on known wheel speeds. By inputting these speeds, the model calculates the robots linear and angular velocities in the global coordinate system. The robots position (x, y) and orientation θ are updated at each time step, and its trajectory is visualized in real time on a 2D plane. This simulation illustrates the robot's omnidirectional mobility resulting from different wheel speeds.
As illustrated in Figure (4.18), two examples of wheel speed configurations are shown:



(a) Wheel speeds: $[0.5; 0.5; 0.5; 0.5]$

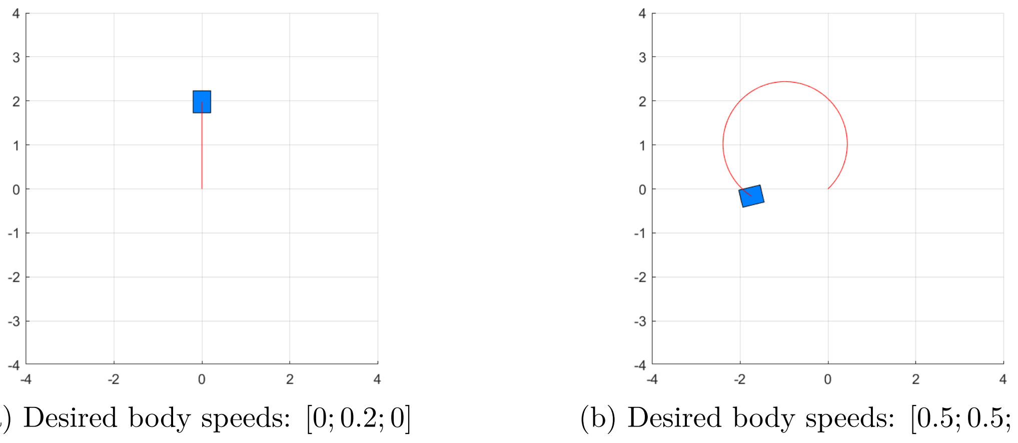


(b) Wheel speeds: $[1; 0; 0; 1]$

Figure 4.18: Forward Kinematics and Path Visualization of a Four-Wheel Mecanum Robot with Wheel Speeds

The second simulation focuses on inverse kinematics. Rather than specifying wheel speeds, this approach starts with a desired velocity, including linear velocities in the x and y directions and an angular velocity around the z-axis. The inverse kinematics model then calculates the wheel speeds required to achieve these desired velocities. The robots position and orientation are updated accordingly, and the resulting path is vi-

sualized similarly to the forward kinematics simulation. This approach emphasizes the precision of inverse kinematics in controlling robot movement based on specific velocity goals.
Figure (4.19) demonstrates two examples of how different desired body speeds affect the resulting wheel speeds for a four-wheeled Mecanum robot:



(a) Desired body speeds: [0; 0.2; 0]   (b) Desired body speeds: [0.5; 0.5; 0.5]

Figure 4.19: Inverse Kinematics and Path Visualization for a Four-Wheel Mecanum Robot with Reference Velocities

Together, these simulations highlight the kinematic capabilities of Mecanum-wheeled robots from different angles. Forward kinematics allows for motion prediction based on wheel speeds, while inverse kinematics determines the wheel speeds needed for a desired motion. These methods complement each other in robot control systems, showcasing the Mecanum wheels versatility and precision in providing omnidirectional movement.

4.4 Simulation of Direct Command Control for Mobile Robotics

This simulation controls a mobile robot by adjusting the speed and direction of its motors to determine the robot's overall speed and position. It incorporates real-world factors like motor speed and direction, with added perturbations to simulate disturbances such as friction or uneven surfaces.
Int the Figure (4.20) we can see all the simulink blocks used for the simulation.

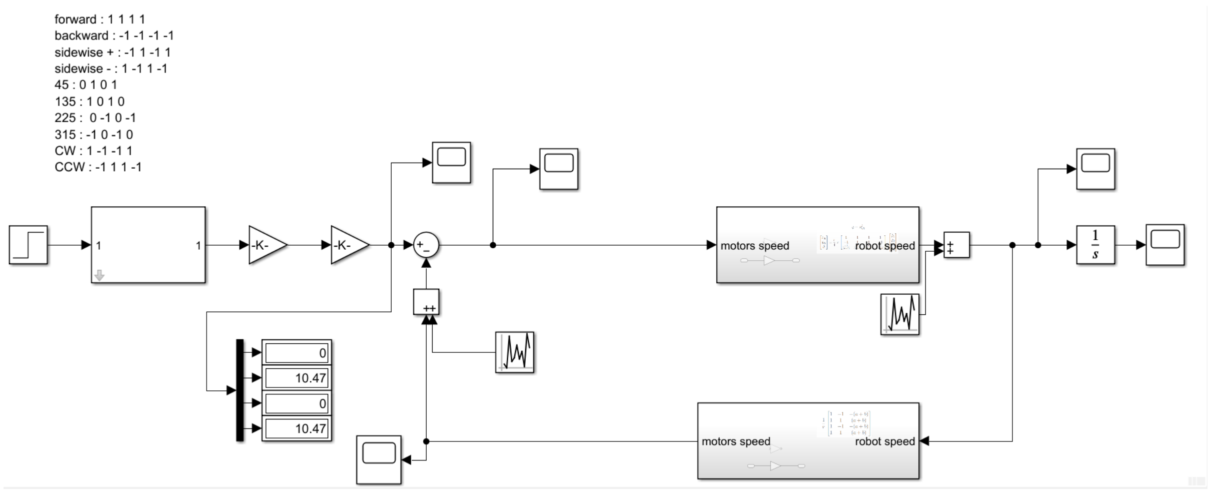
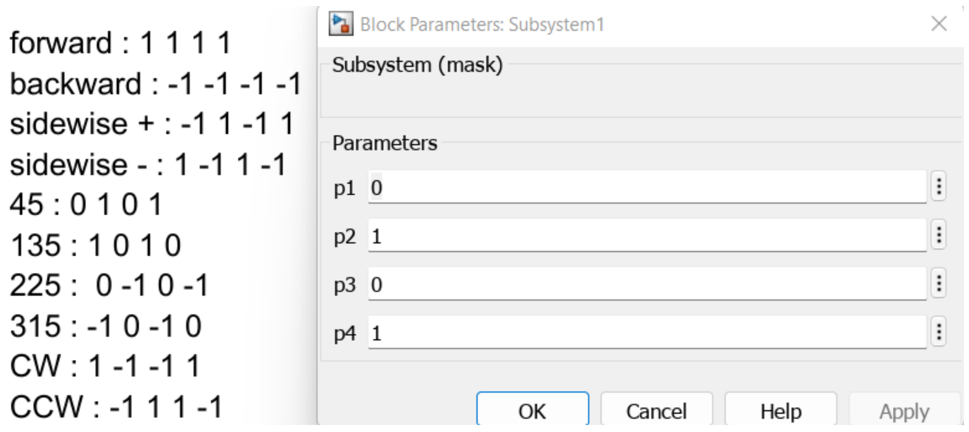


Figure 4.20: Simulation of Direct Command Control

4.4.1 Key Components

- **Motor Speed and Direction Control**: The simulation uses motor speed inputs and directional commands (forward, sideways, diagonal, or rotational) to control the robots movement. Each motor follows specific commands to achieve the desired motion.
  In the Figure (4.21) we can see the interface used to insert the motors speed and directions.



forward : 1 1 1 1
backward : -1 -1 -1 -1
sidewise + : -1 1 -1 1
sidewise - : 1 -1 1 -1
45 : 0 1 0 1
135 : 1 0 1 0
225 : 0 -1 0 -1
315 : -1 0 -1 0
CW : 1 -1 -1 1
CCW : -1 1 1 -1

Block Parameters: Subsystem1

Subsystem (mask)

Parameters

p1	0
p2	1
p3	0
p4	1

Figure 4.21: Input of Desired Speed Parameters

- **Real-world Conventions**: To accurately reflect real-world behavior:
  - **Wheel Diameter**: The robots speed is calculated based on the wheel diameter. Larger wheels result in higher speeds for the same rotational velocity.
  - **Rad/deg Conversion**: Motor control is performed in radians per second, but commands are often given in degrees, requiring conversion.

4.4.2 Simulation Features

- **Perturbations**: Unexpected speed changes simulate real-world disturbances, testing the robustness of the control algorithms and the robots ability to stay on course.
- **Simplified Dynamics**: This simulation focuses on controlling motor speeds and directions without detailed multibody dynamics.

4.4.3 Manual Command Process

The simulation replicates a manual control system (HMI), allowing the operator to issue direct movement commands:

- **Forward**: All wheels move at the same speed.
- **Sideways**: Wheels are adjusted for lateral movement.
- **Rotation (CW/CCW)**: Opposite wheel directions rotate the robot.

The simulation calculates the robots speed and position based on these commands, while also showing how the robot reacts to any disturbances during the process but it is not automated because it needs a control system which made us to make the robot position control simulation.

4.5 Simulation of Position Control for Mecanum-Wheeled Mobile Robots

In this simulation of a mecanum-wheeled mobile robots position control using Simulink and Simscape Multibody, we focus on controlling the robot's position with a closed-loop control system that integrates various subsystems for accurate and realistic feedback.
In the Figure (4.22) we find the simulation including the Simulink blocks.

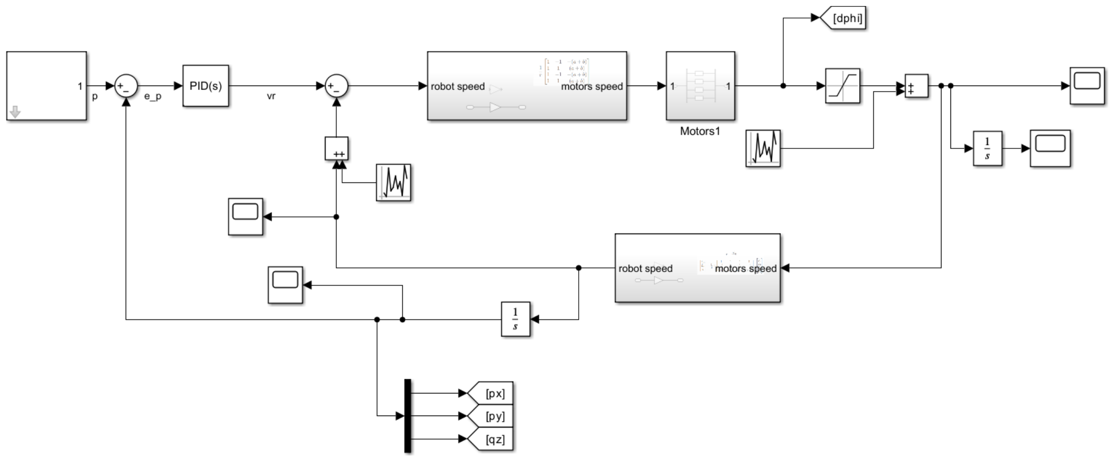
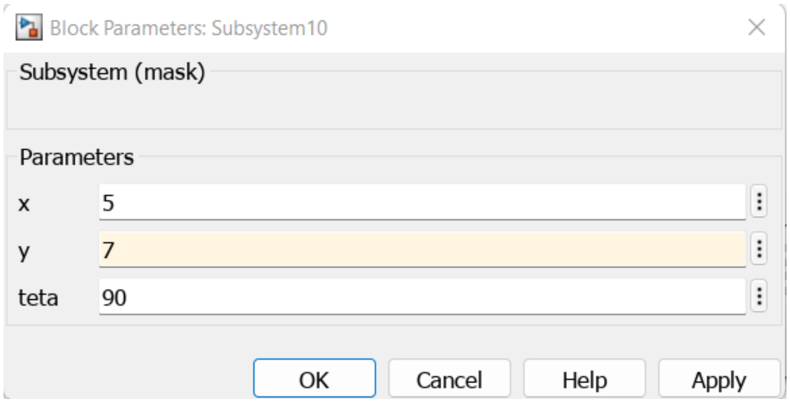


Figure 4.22: Position Control System Architecture of a Mecanum-Wheeled Mobile Robot

4.5.1 Detailed Simulation Setup

Position and Speed Control Loop

PID Controller for Position Control: At the core of the position control is a PD (Proportional-Derivative) controller. The input to this controller is the error between the robot's current position and the desired target position (X, Y, and orientation). This error is then processed to generate the required velocity commands (linear and angular velocities) for the robot.
In the Figure (4.23) we can see the mask used to insert the desired position.


Block Parameters: Subsystem10

Subsystem (mask)

Parameters

x	5
y	7
teta	90

Figure 4.23: Input of Desired Position Parameters

Inverse Kinematics Block: Once the velocity command is calculated, the inverse kinematics block converts the robots required velocities into individual wheel speeds. This is crucial in a mecanum-wheeled robot because each wheel has a different velocity vector to achieve omnidirectional movement. The inverse kinematics equations are implemented as gain matrices to simplify the transformation between robot velocities and motor speeds.

Motor Control Subsystem

Internal Control for Motors: The desired wheel speeds calculated from the inverse kinematics are fed into the motor control block. Each motor has its own system, which ensures the motor's speed tracks the desired speed accurately.
Saturation Block: A saturation block is placed after the motor control to limit the motor speed within a realistic range. This is important because real motors have maximum speed limits that must be respected in the simulation to make it realistic.
Forward Kinematics Block: The actual motor speeds are then converted back into robot velocities using forward kinematics. This ensures that the feedback loop provides the actual velocities, which are used to update the robots position.

Simscape Multibody for Physical Simulation

The Simscape Multibody model represents the physical system of the robot. It consists of:

- **Robot Body and Mecanum Wheels:** The body of the robot is modeled as a rigid solid connected to the wheels using revolute joints. Each wheel is driven by a motor, and its rotation is captured and fed back into the Simulink control system.

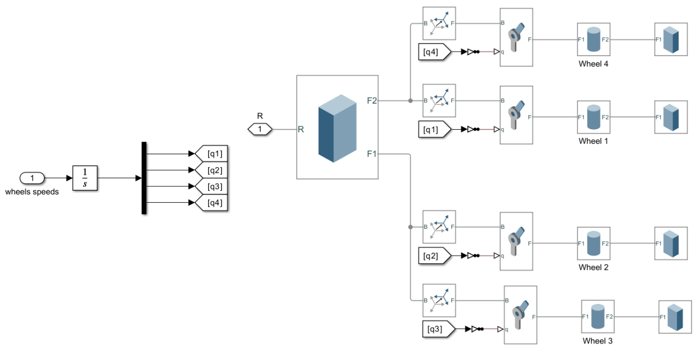


Figure 4.24: 3D Representation of the Robotic System

- **World Reference and Planar Ground:** The robot operates on a planar surface within a defined world reference. This planar surface constrains the robots movement to the XY plane while allowing rotation around the Z-axis (theta).

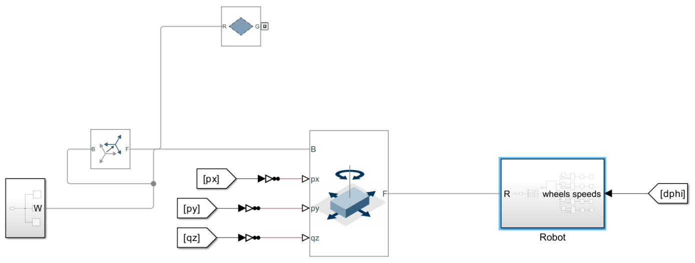


Figure 4.25: Planar and Reference Simulation Overview

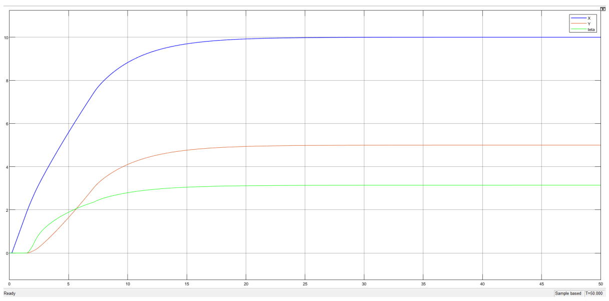
- **Physical Properties:** The wheels and the body are defined with their physical properties such as mass, inertia, and dimensions. These parameters influence the dynamics of the robot in the simulation, ensuring realistic interaction between the robot and the environment.

4.5.2 Simulation Results and Visualization

Scope Values for Speeds and Positions

The simulation provides real-time feedback in the form of speed and position plots using Simulink Scopes.

Position Results (X, Y, Theta): The scope displays the actual position (X, Y) and orientation (Theta) of the robot. As the simulation progresses, these values change, indicating how the robot approaches the desired final position. The scope allows us to see how well the PD controller manages the position error, showing convergence toward the target position over time.
Speed Results (Vx, Vy, Omega): Another scope displays the robot's linear velocities in the X and Y directions (Vx, Vy) and the angular velocity (Omega). These speed values are obtained from the forward kinematics and reflect the real-time velocity of the robot. The graph shows how the robot accelerates and decelerates as it adjusts its movement toward the target.
in the next Figure (4.26) we observe an example of the results captured by the position and speed scoops.


(a) Variation in Position

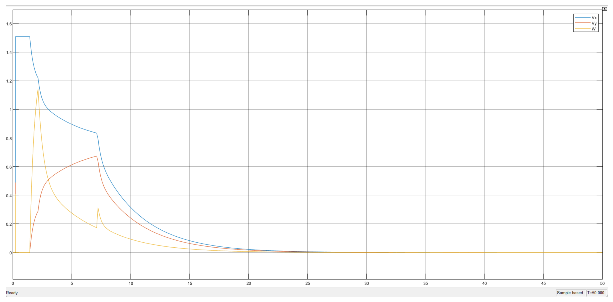

(b) Variation in Speed

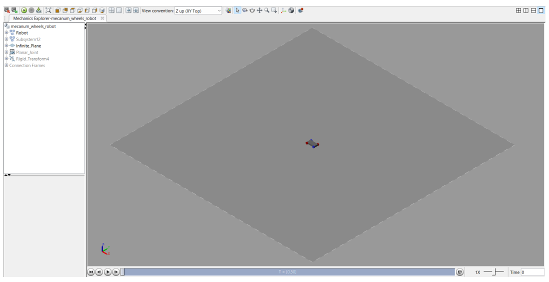
Figure 4.26: Scoop Results Analysis Example

Perturbation and Feedback Behavior

During the simulation, perturbations are introduced, which cause temporary deviations in the robots speed and position. The controllers response to these perturbations is also captured by the scopes, showing how quickly and effectively the system corrects these disturbances.

Simulation Visualization

The simulations mechanical visualization using Simscape Multibody provides a 3D view of the robot as it moves across the plane. This visualization shows how the robot's body and wheels interact with the environment, providing a clear visual representation of the robot's movement.
Position Variation in the Visualization: As the robot moves from its initial position to the target, the visualization updates in real-time, showing the changes in the robots orientation and trajectory. This visual feedback confirms the results observed in the scopes, demonstrating that the robot's position changes according to the input commands and the feedback from the system. In the following Figure (4.27) we can observe the position changement of the 3D model vehicle between it's initial and final position.


(a) Starting Position

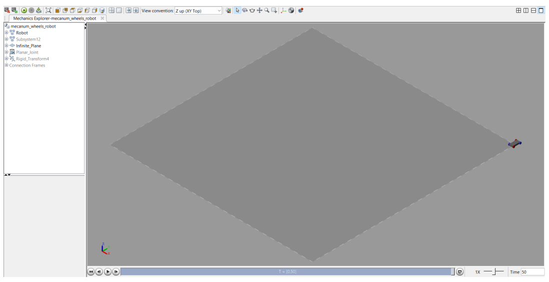

(b) Ending Position

Figure 4.27: 3D Model Visualization

4.5.3 Observations

Convergence to Final Position: The position control system ensures that the robot converges toward the final desired position with minimal overshoot and steady-state error. The PD controller plays a critical role in ensuring smooth transitions, especially during the deceleration phase when the robot approaches the target.
Velocity Profile: The velocity profiles in the scope show how the robot accelerates and decelerates in response to position errors. The control system prevents large spikes in velocity, ensuring that the robot moves smoothly without changing the speed brutally.
Perturbation Response: When perturbations are introduced, the system responds quickly, correcting any deviations in speed or position. This demonstrates the robustness of the control loop, which uses feedback from the physical model to maintain control even in the presence of disturbances.

4.5.4 Conclusion

The simulation of the mecanum-wheeled robot's position control using Simulink and Simscape Multibody provides a highly realistic model for testing and validating control strategies. The combination of a PD controller, kinematic transformations, motor control with feedback, and a detailed physical model ensures that the simulation closely replicates real-world robot behavior. The results observed from the scopes and the visual simulation confirm that the control system effectively manages the robot's movement, compensating for disturbances while maintaining accurate trajectory control.

Chapter 5

Implementation Aspects

Python and Arduino play complementary roles in the field of robotics, leveraging their unique strengths to create sophisticated and efficient robotic systems. Python, known for its readability and extensive libraries, is ideal for scripting, automation, machine learning, and data analysis. It enables complex computations, control algorithms, and advanced functionalities such as computer vision and autonomous navigation, thanks to libraries like OpenCV and TensorFlow. On the other hand, Arduino is a popular open-source electronics platform that provides an easy-to-use micro-controller board and development environment. It excels in prototyping, controlling hardware components, and interfacing with various sensors and actuators. By integrating Python and Arduino, developers can harness the computational power of Python for high-level processing and decision-making while utilizing Arduino's capabilities for real-time control and hardware interfacing. This integration often involves serial communication, where Python scripts send commands to and receive data from Arduino, allowing for tasks such as sensor data acquisition, actuator control, and implementing advanced robotic behaviors. Together, Python and Arduino empower developers to build responsive, intelligent, and versatile robotic systems suitable for a wide range of applications.

5.1 Raspberry Pi

Raspberry Pi is a versatile and compact single-board computer that has captured the imagination of hobbyists, educators, and professionals alike. Its small size and low cost make it accessible to a wide range of users, from beginners learning the basics of programming to advanced developers creating complex projects. Powered by various versions of the Linux operating system, Raspberry Pi can be programmed in multiple languages, including Python, C/C++, and Java, among others. Its GPIO (General Purpose Input/Output) pins allow users to connect sensors, motors, and other hardware components, turning it into a powerful tool for prototyping and experimenting with electronics. Raspberry Pi offers an affordable and versatile platform for realizing creative visions, whether individuals are developing a home automation setup, a multimedia hub, or a robotic project.

Figure 5.1: Raspberry Pi Key Platform for Robotics and Embedded Systems

5.2 Virtual Network Computing

Virtual Network Computing (VNC) is a technology that allows remote access and control of computers over a network. It works by transmitting the graphical desktop environment of the remote system to the viewer, which can be running on a different computer or device. This enables users to interact with and operate a remote computer as if they were sitting in front of it, regardless of physical distance. VNC is widely used for remote technical support, system administration, and accessing files or applications from afar. It provides a convenient solution for collaboration, troubleshooting, and accessing resources across distributed environments.

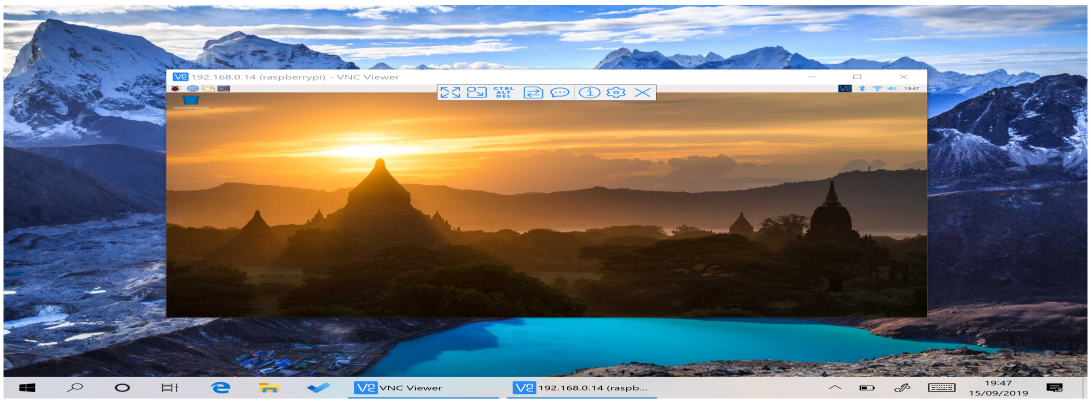


Figure 5.2: Virtual Network Computing Facilitating Remote Access in Robotics and Embedded Systems

RealVNC is a versatile remote desktop software solution that enables users to access and control computers or devices from anywhere in the world. With RealVNC, users can remotely view the desktop interface, transfer files, and even operate applications as if they were physically present at the machine. This software is particularly popular among businesses for its secure and reliable remote access capabilities, facilitating seamless collaboration and troubleshooting across teams. When integrated with Raspberry Pi, RealVNC extends its functionality to the widely used single-board computer, allowing users to remotely manage their Raspberry Pi projects with ease. This combination empowers enthusiasts, hobbyists, and professionals alike to harness the power of remote computing for various applications, ranging from home automation to industrial IoT deployments.

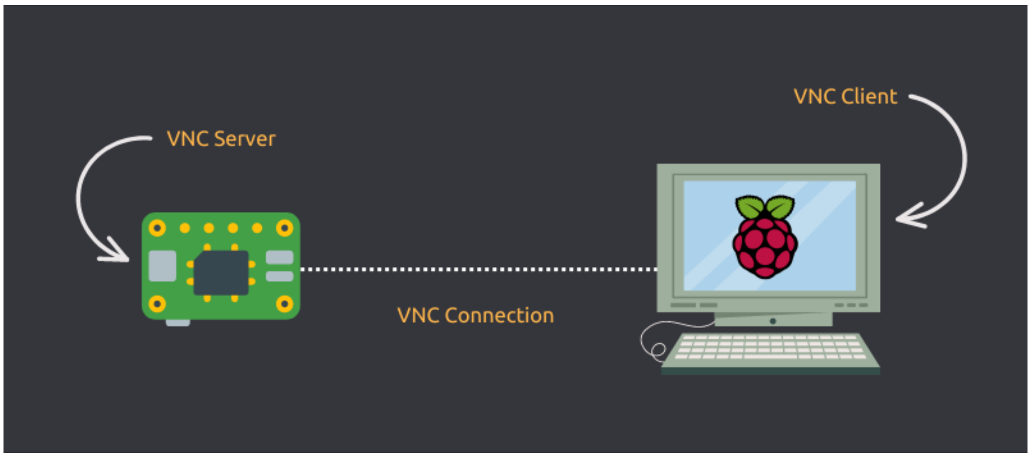


Figure 5.3: Controlling Raspberry Pi With Vnc

5.3 Arduino

Arduino programming is a versatile and accessible way to interact with the physical world through code. Utilizing the Arduino IDE (Integrated Development Environment), developers can write, compile, and upload code to Arduino micro-controller boards, which then execute the programmed instructions. This programming environment is based on a simplified version of C/C++, making it approachable for beginners while still offering advanced functionality for experienced users. With a vast array of sensors, actuators, and shields available, Arduino projects span from simple blinking LED experiments to complex robotics and IoT (Internet of Things) applications.

Figure 5.4: Arduino: The Heart of Creative Electronics Design

5.4 Communication Protocols and Techniques

Raspberry Pi and Arduino can communicate with each other using various communication protocols, enabling them to exchange data and work together in projects. Here's a brief overview of each protocol:

5.4.1 Serial Communication

- Both Raspberry Pi and Arduino support serial communication via UART (Universal Asynchronous Receiver-Transmitter) hardware.
- They can exchange data through RX (Receive) and TX (Transmit) pins using a serial connection.

5.4.2 I2C (Inter-Integrated Circuit)

- I2C is a multi-master, multi-slave serial communication protocol.
- Raspberry Pi and Arduino can communicate as master or slave devices on an I2C bus, facilitating communication between multiple devices.

5.4.3 SPI (Serial Peripheral Interface)

- SPI is a synchronous serial communication interface.
- It enables full-duplex communication between a master device (e.g., Raspberry Pi) and one or more slave devices (e.g., Arduino).

5.4.4 USB Communication

- Raspberry Pi and Arduino can communicate over USB using virtual serial ports.
- This method is convenient for establishing communication without additional hardware, especially when physically connected via USB cables.

5.4.5 Network Protocols (e.g., MQTT)

- MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol suitable for network communication.
- Both Raspberry Pi and Arduino can connect to a network and communicate using MQTT, allowing communication over long distances or with remote servers.

5.5 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. Initially developed by Intel, it has become a key resource for real-time computer vision applications. OpenCV offers a wide range of functions and tools that allow developers to process images and videos to detect and recognize faces, objects, and text. It supports multiple programming languages, including C++, Python, and Java, and can be deployed on various platforms like Windows, Linux, and macOS, as well as on mobile operating systems such as Android and iOS. The library is highly optimized for performance, making it suitable for both research and production purposes. Its extensive suite of algorithms covers numerous applications in robotics, surveillance, gesture recognition, and more, enabling the development of innovative and intelligent vision-based solutions.

Figure 5.5: OpenCV: The Visionary Library for Image Processing

5.6 Graphical User Interface

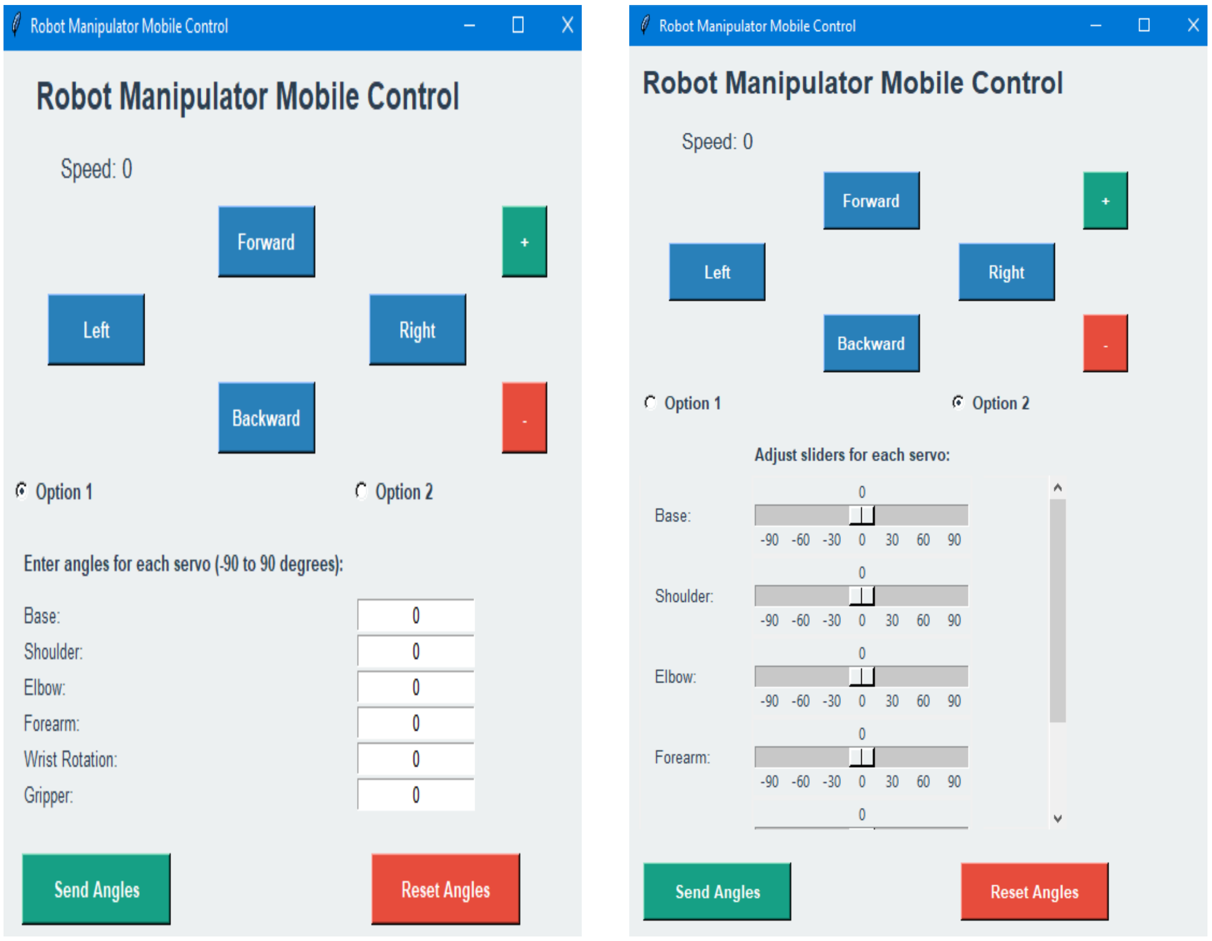
A graphical user interface (GUI) is a visual platform that allows users to interact with electronic devices using graphical elements such as icons, buttons, and menus, rather than text-based interfaces or command-line prompts. GUIs are designed to be intuitive and user-friendly, enabling users to perform tasks through direct manipulation of these visual elements. This approach to interface design enhances accessibility and efficiency, making it easier for people with varying levels of technical expertise to use computers and other digital devices [20].
In Python, the Tkinter library is a popular choice for creating graphical user interfaces. Tkinter is the standard GUI toolkit for Python, providing a simple way to create windows, dialogs, buttons, menus, and other interactive components [21].

Figure 5.6: Tkinter - GUI for Python

5.6.1 Graphical User Interface for Robot Manipulator Mobile Control

In this study, we developed a graphical user interface (GUI) using the Tkinter library in Python to facilitate the control and monitoring of robotic systems. The primary objective of the GUI is to provide an intuitive and user-friendly platform for interacting with various robotic components. The interface includes multiple widgets such as buttons, sliders, and text fields, which allow users to input commands and receive real-time feedback from the robots. The design of the GUI emphasizes modularity and scalability, enabling easy integration with different robotic platforms and sensors.

The development of a Graphical User Interface (GUI) for a robotic manipulator using Python's Tkinter library provides an efficient means of controlling a robotic system. This GUI allows users to manage both the movement of a mobile robot and the precise control of a robotic arm with six degrees of freedom. The interface includes directional buttons for movement (forward, backward, left, right) and speed control buttons to adjust the robot's speed within a defined range. It features two input methods for adjusting the robotic arms angles: entry fields, which accept angles between -90 and 90 degrees, and sliders, which offer a more interactive approach. Users can switch between these input methods using radio buttons, with the GUI dynamically updating to reflect the selected option. Real-time feedback on angle settings and speed is provided, and users have the capability to reset all angles to zero and stop the robot's movements. The interface also integrates keyboard controls for directional movement and speed adjustment, ensuring a responsive and user-friendly experience. This Tkinter-based GUI exemplifies a practical application in robotics, combining functionality with ease of use to enhance both operational and educational interactions with robotic systems.



(a) Option 1 (b) Option 2

Figure 5.7: Graphical User Interface for Controlling a Mobile Robot Manipulator

The flowchart presented outlines the systematic process for managing a robot via a graphical user interface (GUI). It starts with initializing the GUI and binding key events to enable user interaction. The flowchart then details the selection between two input methods: entry fields for direct angle input or sliders for real-time adjustments. Following input selection, it describes the steps for validating and applying servo angles when the 'Start' button is pressed. It also includes mechanisms for adjusting speed, both increasing and decreasing, and handling cases where the speed remains unchanged. The flowchart further addresses directional controls, detailing how the robot responds to key presses for movement or stopping. Finally, it covers the reset functionality, which returns the robot and angles to their default settings. This structured representation ensures a clear understanding of the operational workflow and user interactions involved in robot control through the GUI, as shown in Figure (5.8).

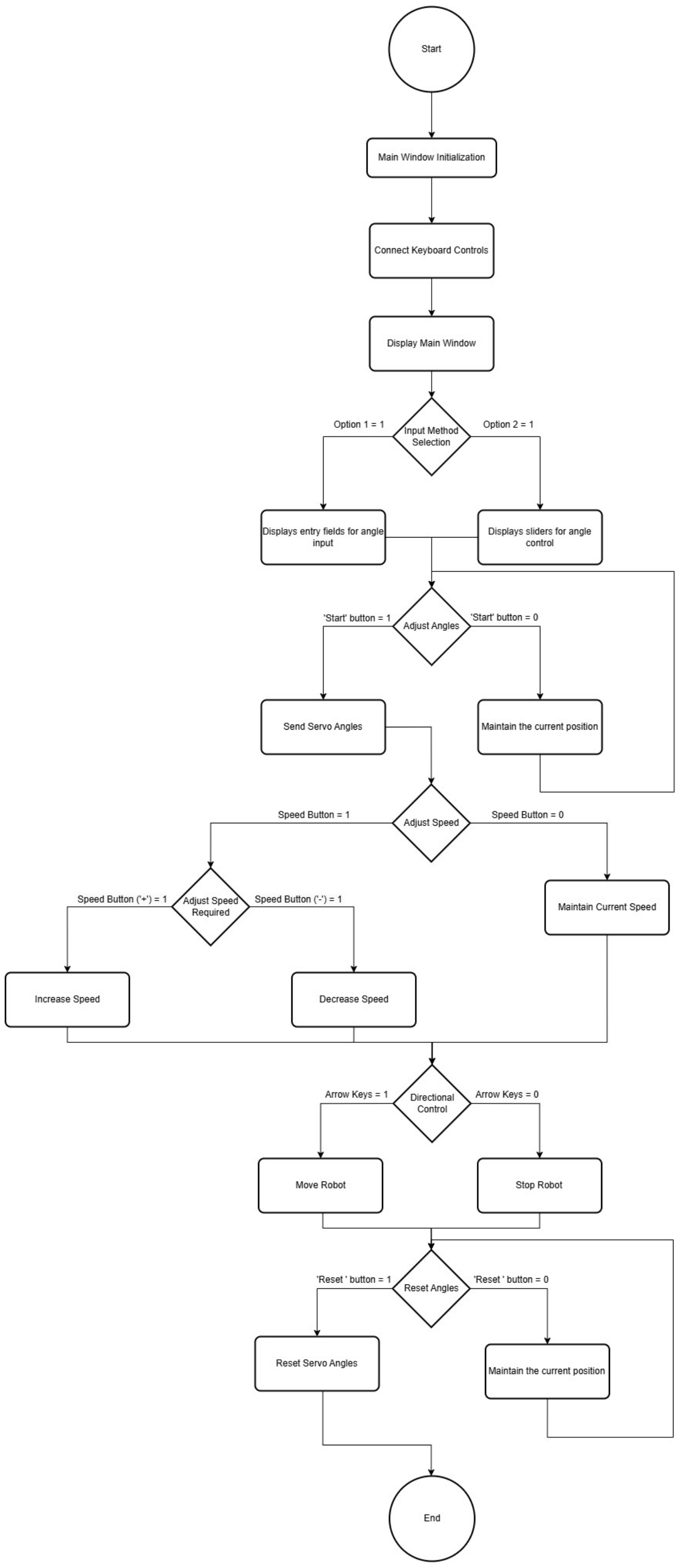
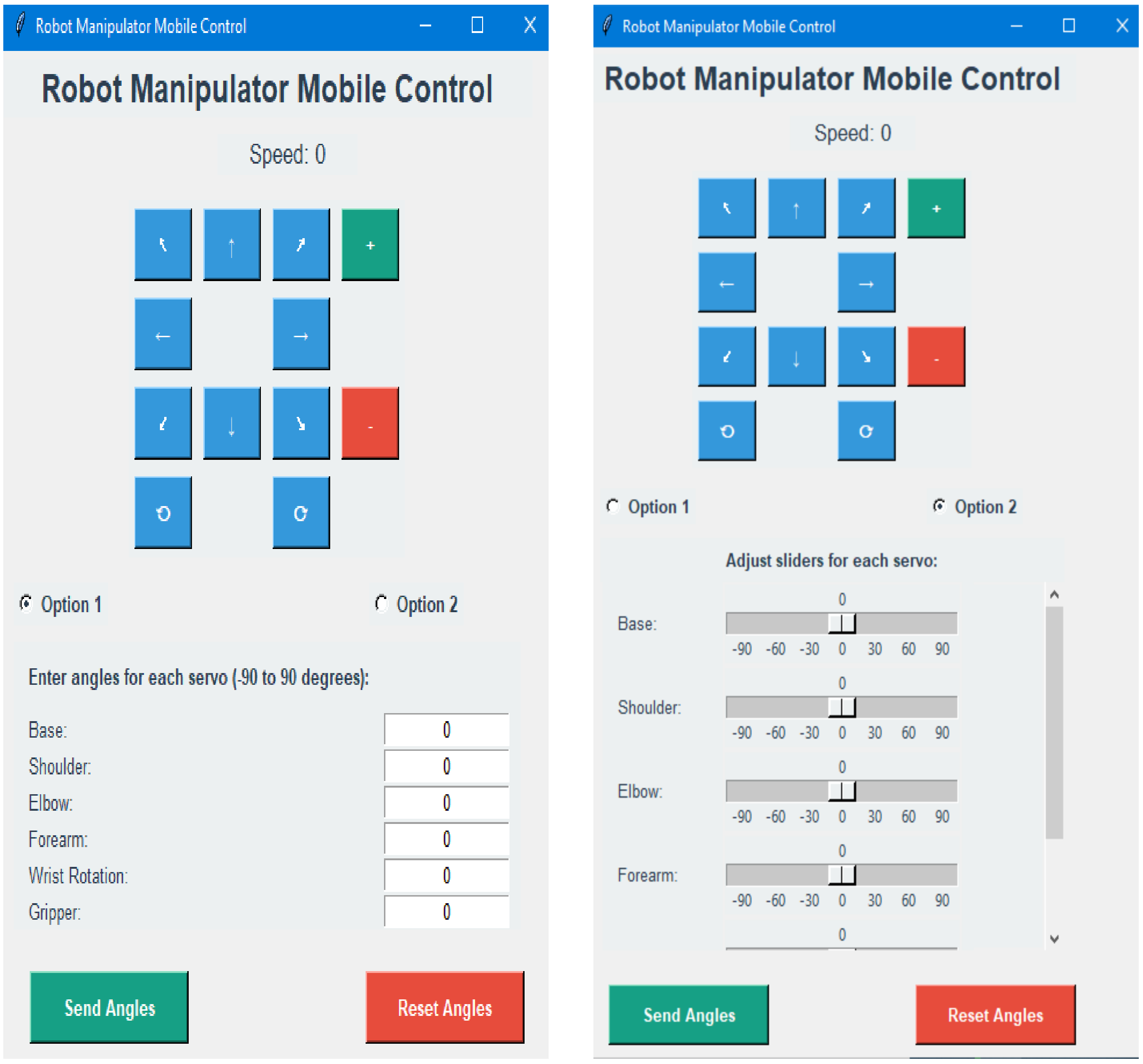


Figure 5.8: This Figure presents a detailed flowchart outlining the sequential process for controlling a robot through a graphical user interface (GUI). It includes the components for speed adjustments, directional movements and management of servo angles.

The integration of Mecanum wheels in robotic systems enables advanced mobility, allowing omnidirectional movement such as rotation and diagonal motion. This expanded capability is reflected in the updated Tkinter GUI, which now includes buttons for additional movements: Rotate Left, Rotate Right, and various diagonal directions. Each button is mapped to a corresponding function in the code, providing intuitive control over the robot's movement. The GUI layout has been optimized for consistent button size and spacing, ensuring ease of use and enhancing the overall user experience in controlling the robot.


Robot Manipulator Mobile Control
Robot Manipulator Mobile Control
↖ ↑ ↗ +
← →
↙ ↓ ↘ -
Send Angles
Reset Angles
Robot Manipulator Mobile Control
Robot Manipulator Mobile Control
↖ ↑ ↗ +
← →
↙ ↓ ↘ -
-90 -60 -30 0 30 60 90
Send Angles
Reset Angles

Speed: 0

Option 1 [✓] Option 2

Enter angles for each servo (-90 to 90 degrees):

Base:	0
Shoulder:	0
Elbow:	0
Forearm:	0
Wrist Rotation:	0
Gripper:	0

Speed: 0

Option 1 [✓ Option 2]

Adjust sliders for each servo:

Base:	0
Shoulder:	0
Elbow:	0
Forearm:	0

(a) Option 1 (b) Option 2

Figure 5.9: Graphical User Interface for Controlling an All-Directional Mobile Manipulator Robot

5.6.2 Graphical User Interface for Pathfinding Visualization

The Pathfinding Visualizer GUI, created using Python's Tkinter library, provides an interactive platform for exploring and comparing the A* and Dijkstra algorithms in a grid-based environment. The interface allows users to configure start and goal positions, as well as place obstacles, by clicking on the grid cells. Radio buttons enable easy selection between placing start points, goal points, or obstacles, with the grid dynamically updating to reflect these changes. Clear color coding is used throughout the interface, with the start position marked in green, the goal in red, and obstacles in black, ensuring easy distinction of these elements.
Users can select either A* or Dijkstras algorithmor a combination of both via radio buttons, and a **"Start"** button initiates the selected algorithm(s). The paths generated by each algorithm are displayed in real-time, with A* paths visualized in blue and Dijkstra paths in purple, allowing users to see the differences in how the algorithms navigate the grid. The **"Reset"** button provides the ability to clear the grid and start fresh, encouraging experimentation with different setups.
The GUI features a legend to explain color codes for start and goal positions, obstacles, and paths, making it easy for users to interpret the visual output. It allows simultaneous visualization of both algorithms, providing a side-by-side comparison to highlight their different pathfinding strategies.

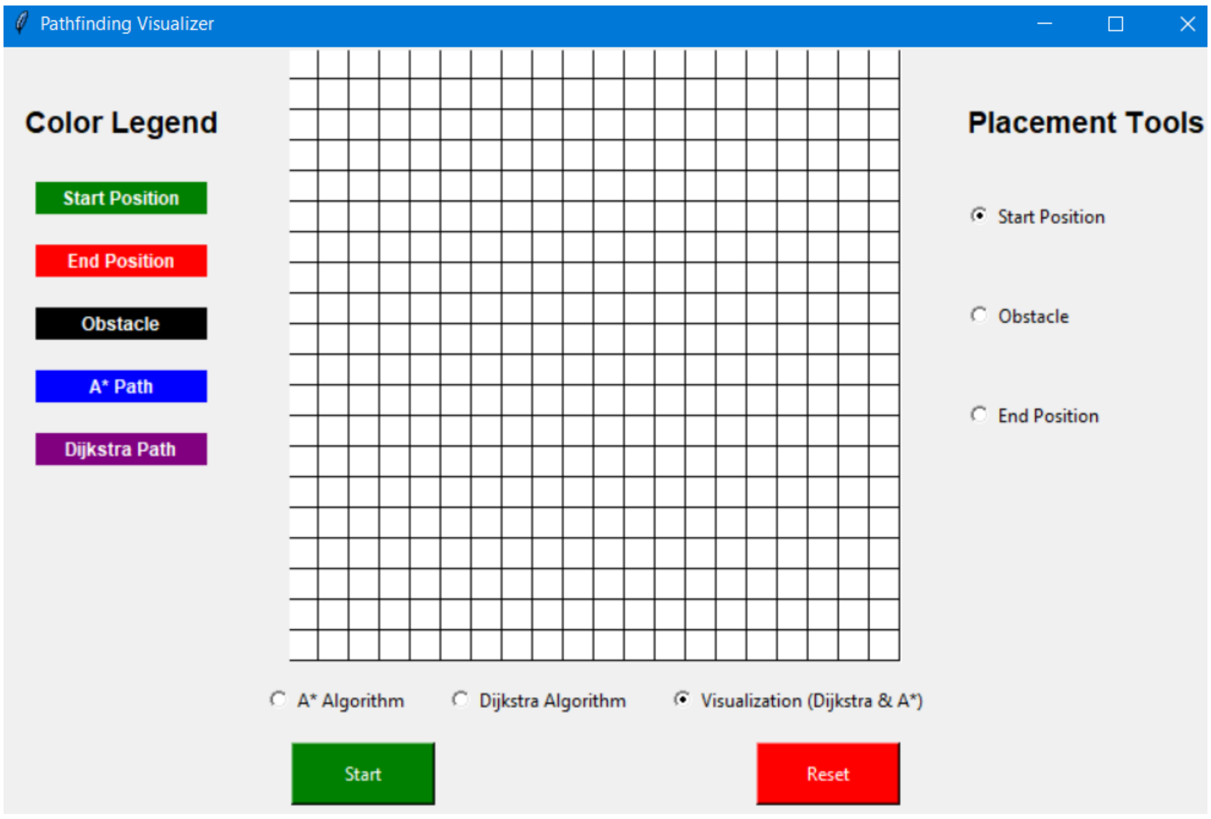


Figure 5.10: Development and Implementation of a Pathfinding Visualization Interface Using Tkinter: Analysis of A* and Dijkstra Algorithms

5.6.3 Line Following Status: GUI Interface and Visualization

Integrating a graphical user interface (GUI) with a microcontroller-based control system greatly improves the management and monitoring of a line-following robot, enhancing operational efficiency and user interaction. Developed with Tkinter, the GUI serves as the main interface for real-time status monitoring, featuring color-coded LED indicators to represent various operational modes. These modes include:

- **STOP**: Indicates that the robot halts due to the absence or complete detection of the line.
- **SHARP_LEFT** and **SHARP_RIGHT**: Signal sharp turns in response to detected line curves.
- **LEFT** and **RIGHT**: Represent moderate deviations from the line.
- **FORWARD**: Shows that the robot is moving straight ahead.

The LEDs change color dynamically based on the robot's status, offering immediate visual feedback for operators to assess and adjust the robot's behavior. The GUI is depicted in Figure (5.11).

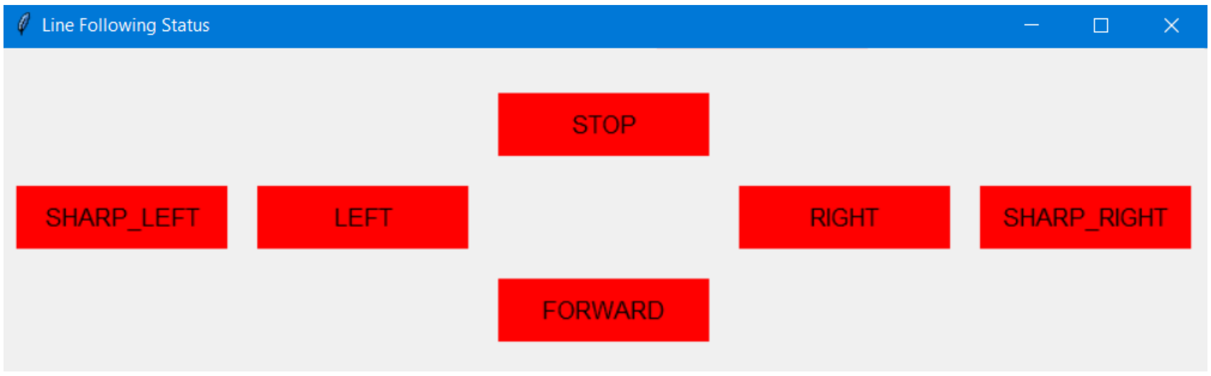


Figure 5.11: Graphical User Interface for line-following robot.

The microcontroller, commonly an Arduino, plays a pivotal role in the line-following robot's control system by processing inputs from line-tracking sensors. These sensors continuously monitor the robots position relative to the line, detecting variations in line presence and direction.
The integration of the microcontroller with a Tkinter-based graphical user interface (GUI) establishes a comprehensive control system for the robot. The GUI displays real-time operational status through color-coded LED indicators, dynamically updated based on data from the Arduino. This setup allows operators to effectively monitor and adjust the robots behavior. Real-time feedback from the GUI enhances user interaction by providing an intuitive view of the robots performance and modes.

5.6.4 Real-Time QR Code Detection System

This real-time QR code detection system integrates advanced computer vision techniques with a user-friendly graphical interface. The system is designed to efficiently capture and process video input from a camera, identifying and decoding QR codes within the live feed. The detection process involves analyzing each frame in real-time, where QR codes are identified, highlighted, and their data is displayed along with the coordinates of their center [22]. This combination of real-time processing and intuitive control makes the system a valuable tool for applications requiring quick and accurate QR code recognition [22].

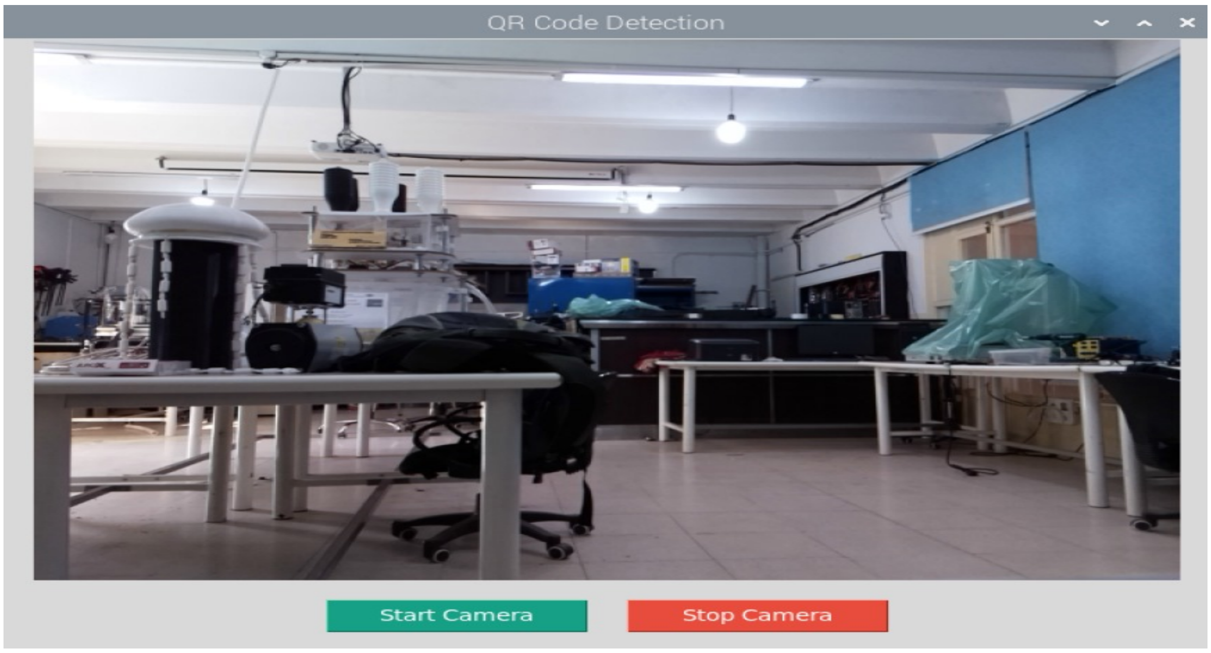


Figure 5.12: GUI Visualization of Real-Time QR Code Detection System

In addition to the real-time QR code detection, the system has been enhanced to estimate the distance between the camera and the detected QR code. This functionality is achieved by applying the pinhole camera model, which utilizes the known real-world width of the QR code and the detected width in pixels. The system calculates the focal length of the camera in pixels based on its horizontal field of view and resolution. By measuring the width of the QR code in the captured image, the program estimates the distance in millimeters, which is then converted to centimeters for easier interpretation. This estimated distance is displayed directly on the video feed, providing users with both the QR code data and the approximate distance from the camera.

5.6.5 Real-Time QR Code Detection and Servo Motor Control Integration

The integration of computer vision and robotic control is demonstrated through a Python-based graphical user interface (GUI) utilizing Tkinter, OpenCV, and PyZbar. This application enables real-time QR code detection and servo motor control via an interactive interface. OpenCV captures live video, while PyZbar decodes QR codes from each frame. Frames are processed in grayscale to improve detection accuracy, and QR codes are identified and analyzed. When the decoded data matches a user-selected QR type (e.g., "Blue," "Red," or "Green"), the system draws a bounding rectangle around the QR code and displays the information on the video feed. The servo motor is then halted as part of the control feedback loop.
The user interface includes buttons to start and stop the video feed and servo motor, along with radio buttons for selecting the QR code type for detection filtering. This modular design allows for interactive control based on camera input. By combining real-time QR code detection with servo control, the system effectively integrates vision-based object recognition with mechanical control, offering a robust solution for automation tasks. A visual representation of this integration is shown in Figure (5.13), which illustrates the real-time QR code detection and servo motor control interface.

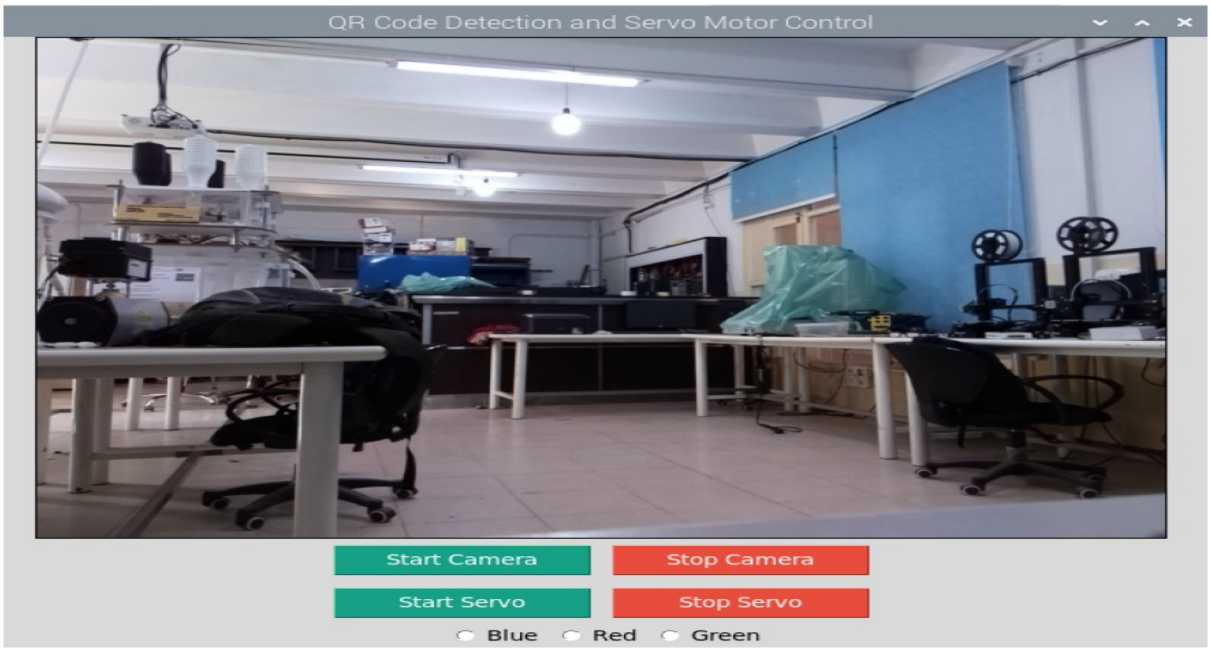


Figure 5.13: GUI for Real-Time QR Code Detection and Servo Motor Control Integration

5.6.6 Tkinter GUI for Managing and Executing Multiple Python Scripts

A graphical user interface (GUI) is developed using Python's Tkinter library to manage and execute multiple scripts. This interface allows users to run various scripts sequentially through a set of interactive buttons. Each button is linked to a specific script, and selecting a button will halt any currently running script before initiating the chosen one. This design ensures that only one script is active at a time, thus avoiding conflicts. Additionally, the GUI features a **"Home"** button that stops any ongoing processes and refreshes the main window, improving user control and navigation. The interface also manages the application's closing event effectively, ensuring all processes are properly terminated before exiting. This setup offers a user-friendly and efficient way to control script execution within a cohesive Tkinter-based environment, as illustrated in Figure (5.14).

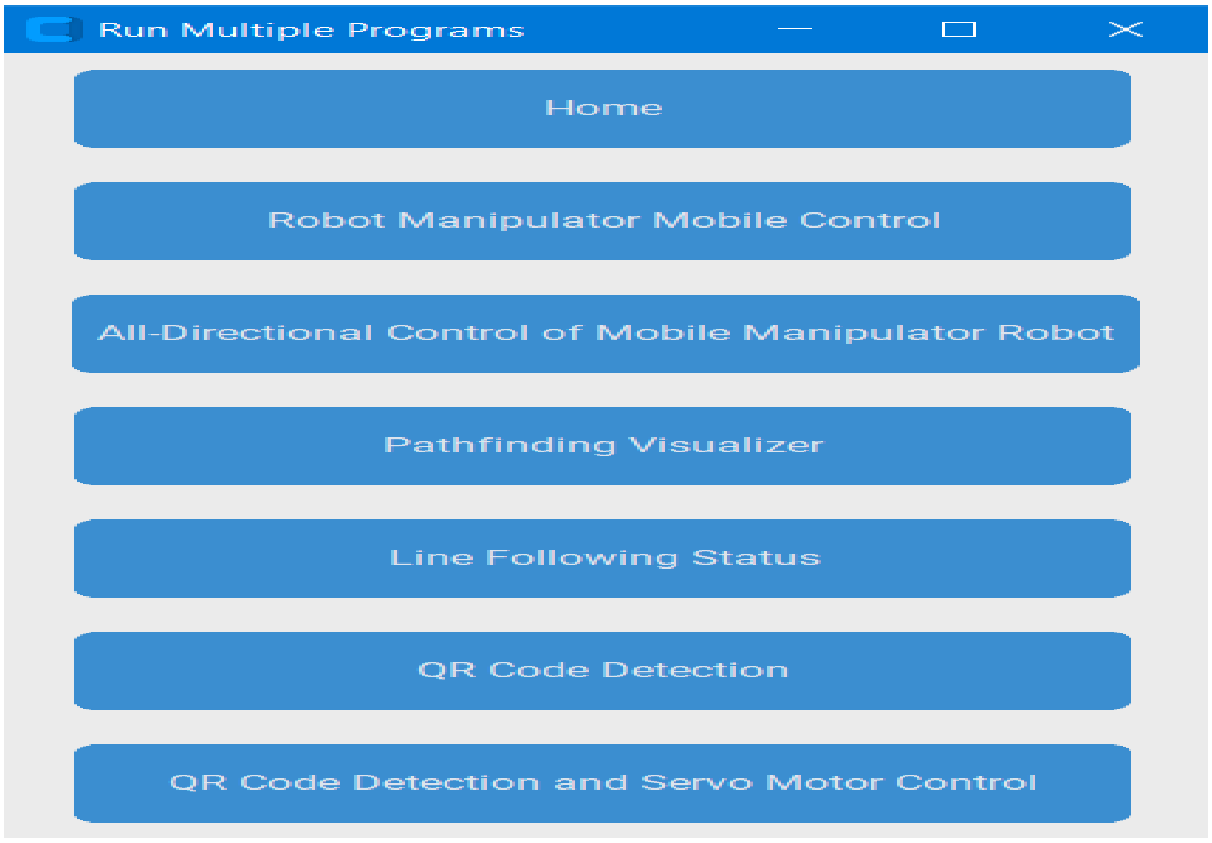

Run Multiple Programs
Home
Robot Manipulator Mobile Control
All-Directional Control of Mobile Manipulator Robot
Pathfinding Visualizer
Line Following Status
QR Code Detection
QR Code Detection and Servo Motor Control

Figure 5.14: Graphical User Interface for Managing and Executing Multiple Scripts.

5.7 Functionality and Features of the Robotics Control Interface

This section has introduced the graphical user interface (GUI) designed for controlling robotic systems, focusing on both mobile robots and robotic arms. The GUI enhances user interaction through intuitive controls for navigation and manipulation.In the initial state, the mobile robot manipulator is positioned at its home configuration, where all joint angles are set to zero. This configuration ensures that the manipulator is in a known and repeatable position, which is crucial for calibration and starting tasks. The base of the robot is stationary, and the manipulator arm is fully extended along a predefined axis.

This initial state serves as a reference point for subsequent movements and operations, allowing for precise control and repeatability in tasks such as pick-and-place.
As shown in Figure (5.15), the manipulator's home configuration aligns its joints along the predefined axis, providing a stable and consistent starting position for task execution.

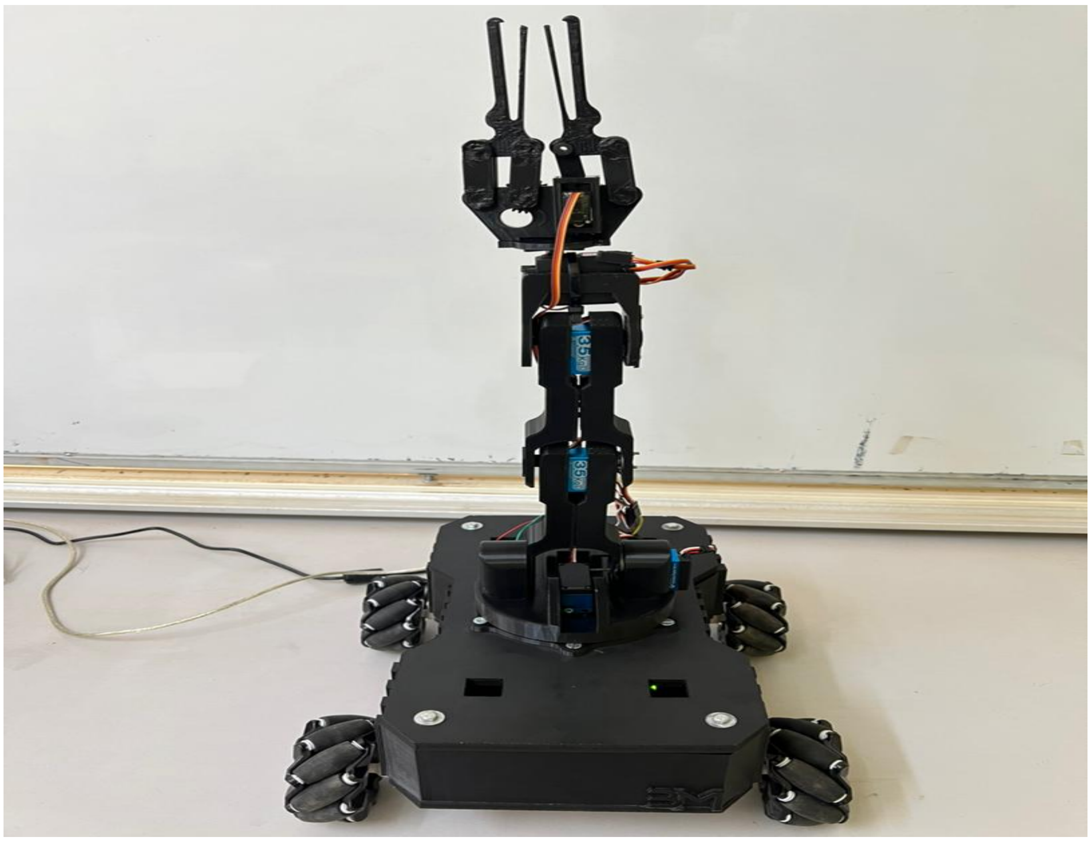
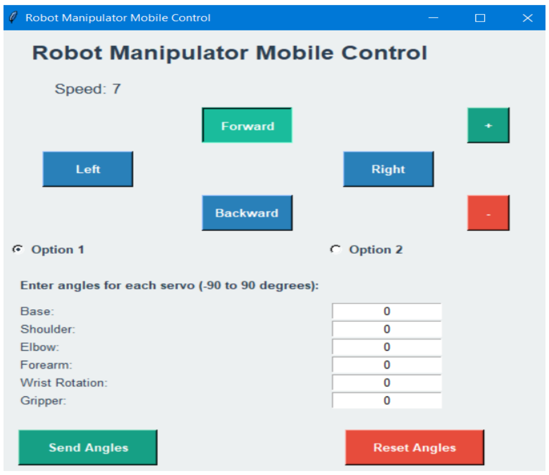


Figure 5.15: Robot Manipulator Mobile in Initial State

5.7.1 Mobile Robot Control GUI

The integration of Mecanum wheels into a mobile robot allows for omnidirectional movement, enhancing its ability to maneuver in various directions. Using a graphical user interface (GUI) and keyboard controls, the robot can be operated by pressing the corresponding arrow keys. For instance, pressing the up arrow moves the robot forward, the down arrow moves it backward, while the left and right arrow keys enable lateral movement and speed adjustment further streamline control. This seamless control provides an intuitive interface for the user, where the robot's movement is displayed in real-time within the GUI, offering both visual feedback and dynamic interaction with the mobile robot, as shown in Figure (5.17) and Figure (5.19).



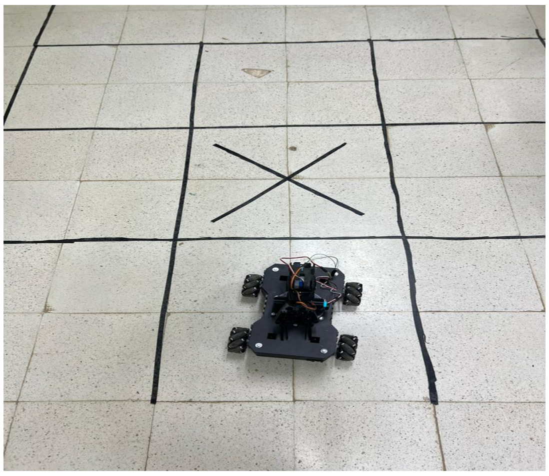
Robot Manipulator Mobile Control

Speed: 7

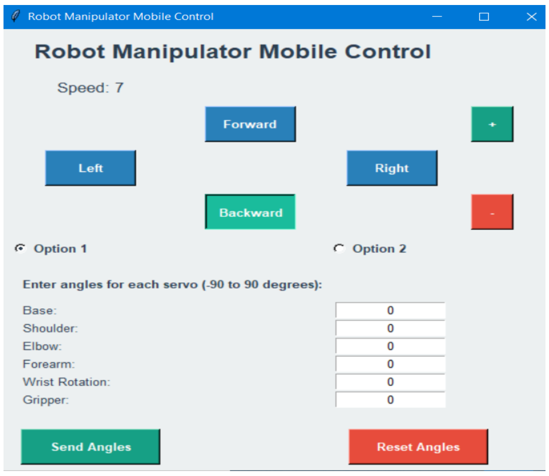
[✓ Option 1] Option 2

Enter angles for each servo (-90 to 90 degrees):

Base:	0
Shoulder:	0
Elbow:	0
Forearm:	0
Wrist Rotation:	0
Gripper:	0



Forward movement control



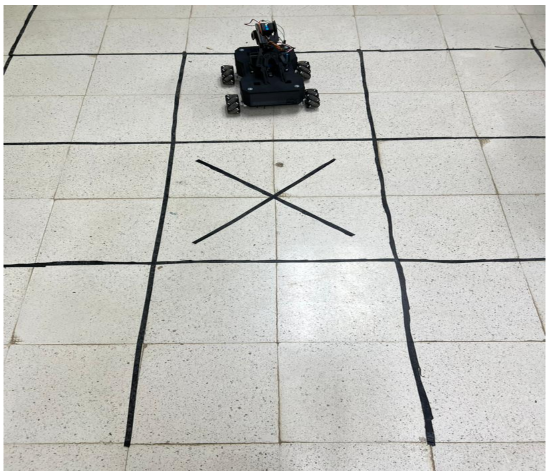
Robot Manipulator Mobile Control

Speed: 7

[✓ Option 1] Option 2

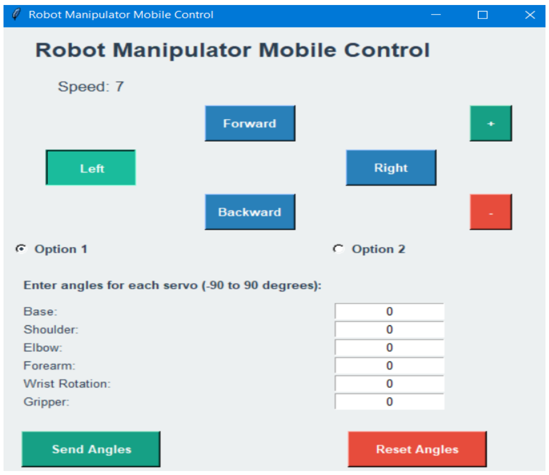
Enter angles for each servo (-90 to 90 degrees):

Base:	0
Shoulder:	0
Elbow:	0
Forearm:	0
Wrist Rotation:	0
Gripper:	0



Backward movement control

Figure 5.17: Results of the GUI Showing Control for Forward and Backward Movements of the Mobile Robot with Mecanum Wheels



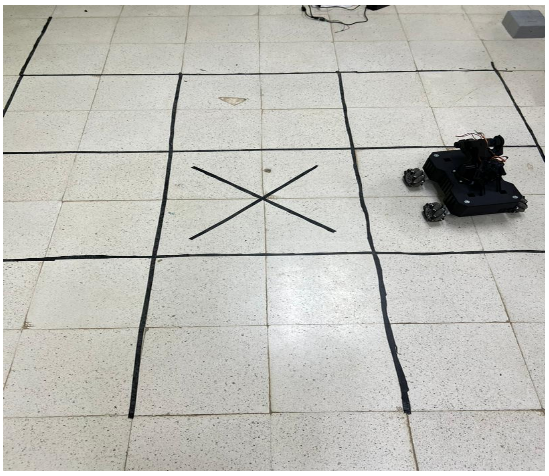
Robot Manipulator Mobile Control

Speed: 7

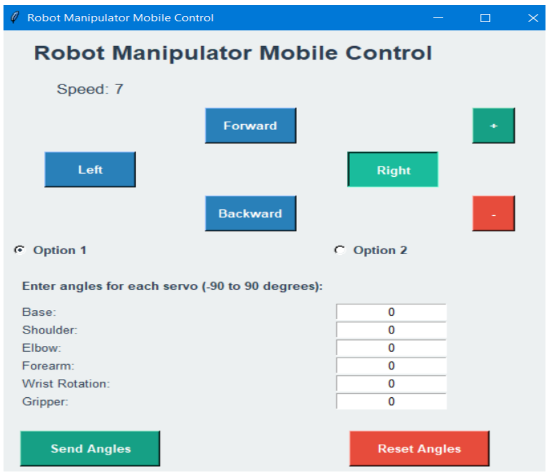
[✓ Option 1] Option 2

Enter angles for each servo (-90 to 90 degrees):

Base: 0
Shoulder: 0
Elbow: 0
Forearm: 0
Wrist Rotation: 0
Gripper: 0



Left movement control



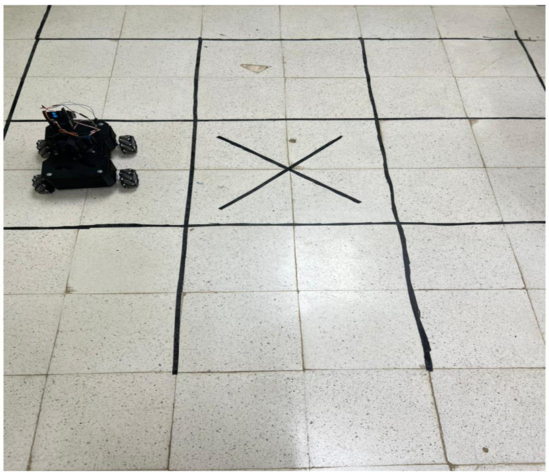
Robot Manipulator Mobile Control

Speed: 7

[✓ Option 1] Option 2

Enter angles for each servo (-90 to 90 degrees):

Base: 0
Shoulder: 0
Elbow: 0
Forearm: 0
Wrist Rotation: 0
Gripper: 0

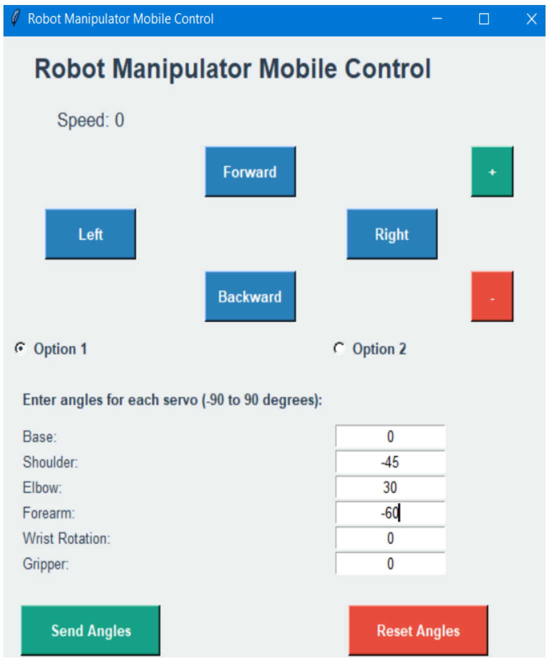


Right movement control

Figure 5.19: Results of the GUI showing control for left and right movements of the mobile robot with Mecanum wheels

5.7.2 Robotic Arm Control GUI

The graphical user interface (GUI) for the robotic arm provides real-time control over the arm's angles. The interface displays the angles for each joint directly on the GUI, allowing the user to visualize different configurations of the arm. For instance, if the shoulder joint is set to -45 degrees, the elbow to 30 degrees, and the forearm to -60 degrees. The result is shown in the following Figure (5.20).


Robot Manipulator Mobile Control

Robot Manipulator Mobile Control

Speed: 0

Forward

+

Left

Right

Backward

-

[✓ Option 1] Option 2

Enter angles for each servo (-90 to 90 degrees):

Base:	0
Shoulder:	-45
Elbow:	30
Forearm:	-60
Wrist Rotation:	0
Gripper:	0

Send Angles

Reset Angles

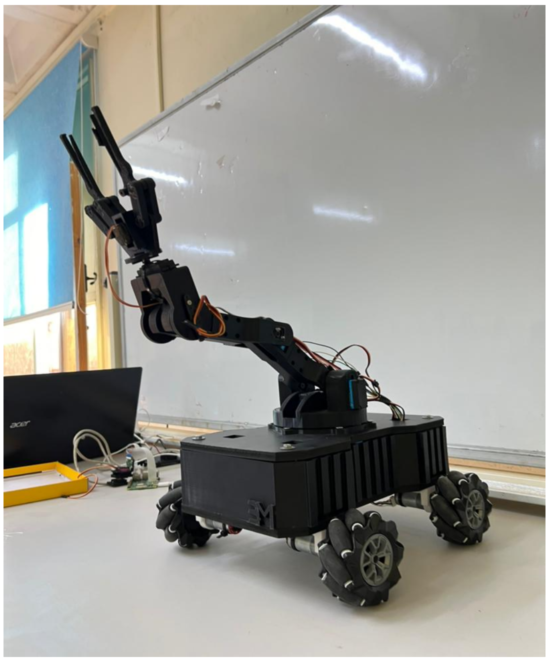
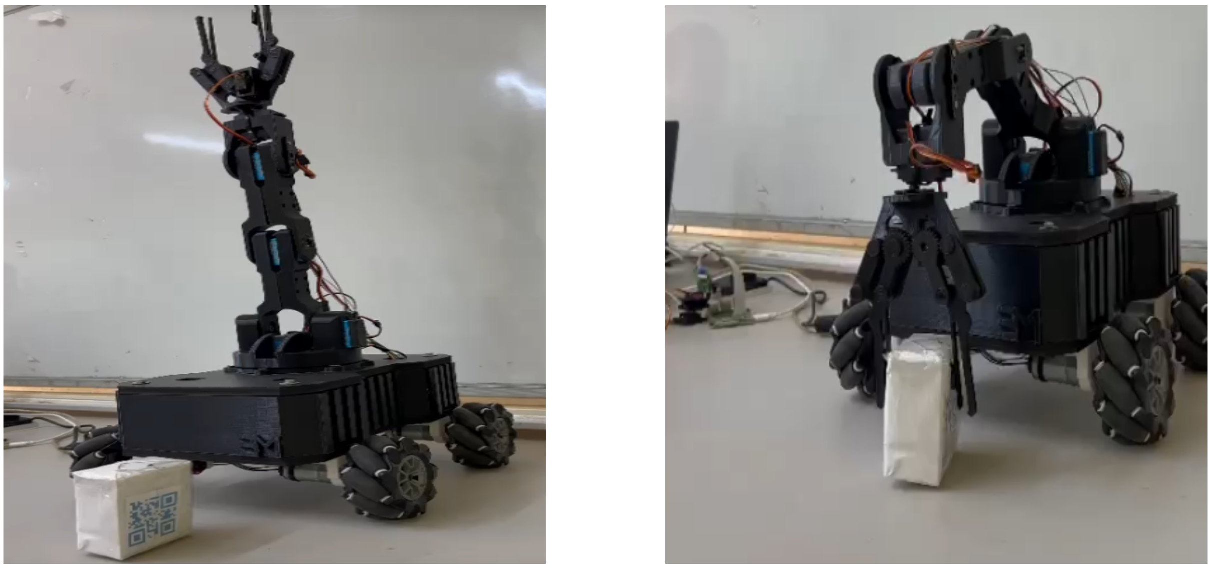
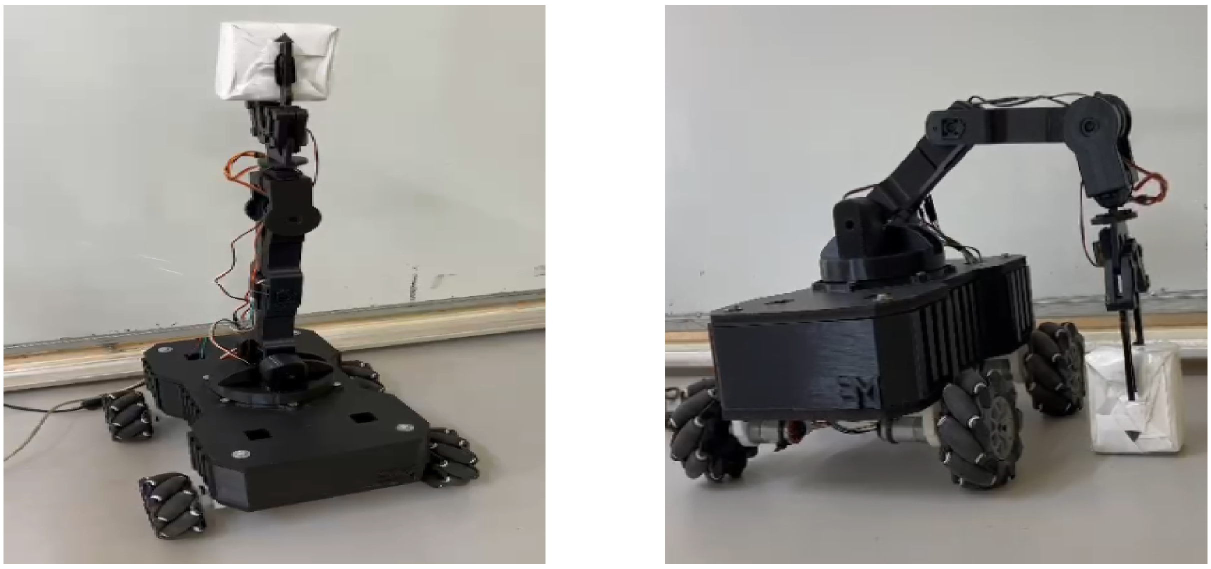


Figure 5.20: Graphical User Interface for the Robotic Arm Showing Real-Time Angle Adjustments

A pick-and-place operation uses a GUI to control the angles of servo motors in a robotic arm. By adjusting these angles, the arm moves to pick an object from one location and place it in another. The GUI allows users to input the servo angles, coordinating the movement of the arm's joints to ensure accurate and smooth positioning of the end effector. This simplifies the process, enabling precise and repeatable pick-and-place tasks. This visualization of the pick-and-place operation, as illustrated in Figure (5.22), demonstrates how the robotic arm adjusts its configuration to handle objects with precision and efficiency.



Pick Operation



Place Operation

Figure 5.22: Graphical user interface for the robotic arm showing real-time angle adjustments

5.7.3 Pathfinding and Robotic Arm Operations

In this study, we explore the implementation of pathfinding algorithms and robotic arm operations within a graphical user interface (GUI) for a mobile robot. The pathfinding visualization employs both the A* and Dijkstra algorithms on a (3x3) grid. The GUI is designed to demonstrate the functionality of these algorithms in navigating the mobile robot from a start position of (0,0) to a goal position of (2,2), considering obstacles located at coordinates (2,0) and (1,2) as shown in Figure (5.23) and Figure (5.24). Additionally, we address the operations of a robotic arm tasked with executing a pick-and-place operation. The robotic arms objective is to transport an object from a start position (0,0) to a goal position (2,2). The operation involves moving the arm to the start position, grasping the object, navigating to the goal position, and then releasing the object. This integrated system highlights the interaction between mobile robot navigation and robotic arm operations in achieving complex objectives.

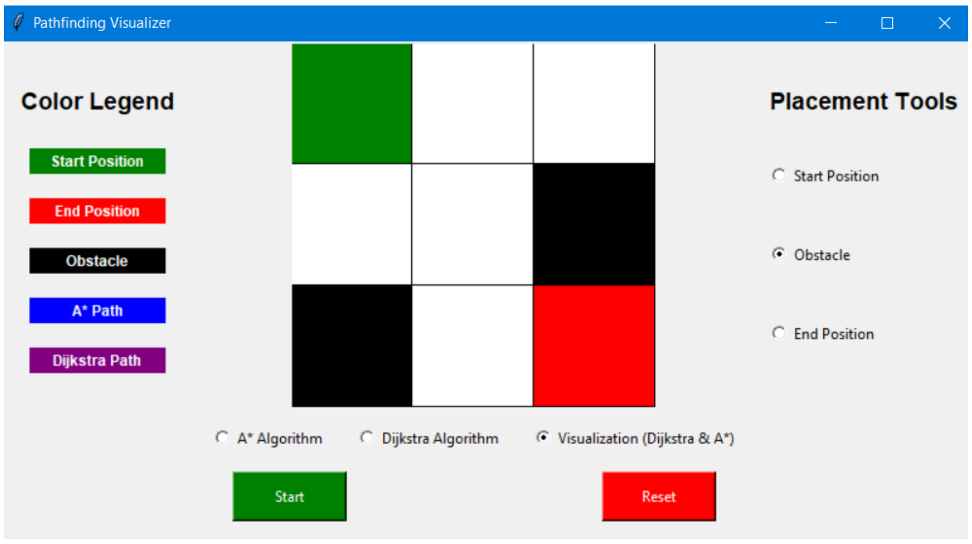


Figure 5.23: Pathfinding visualization GUI for the mobile robot

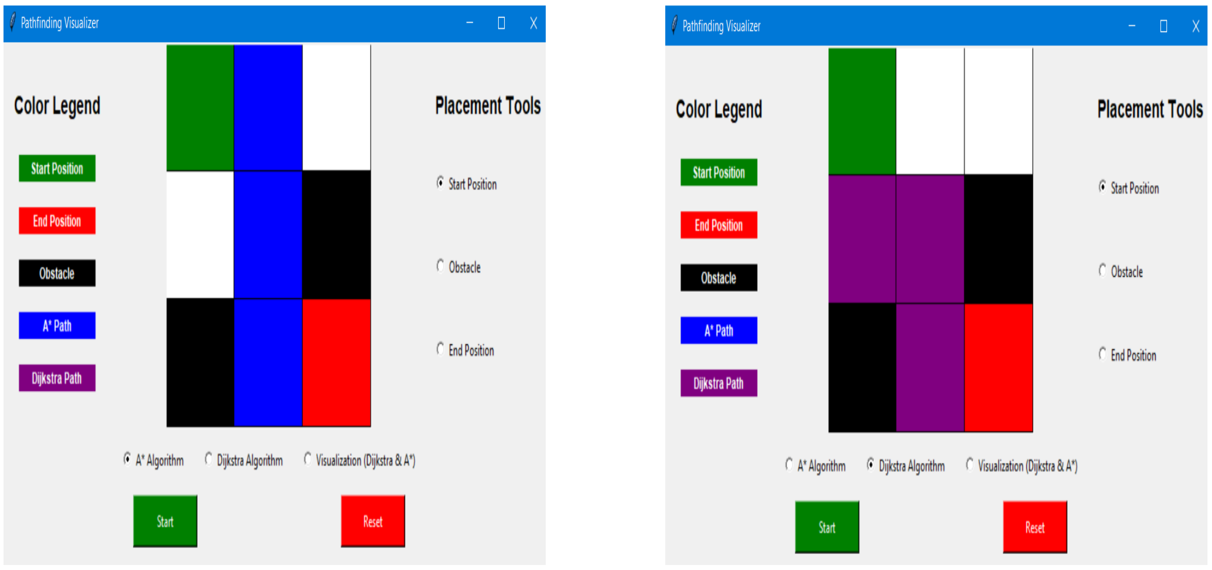


Figure 5.24: Pathfinding Visualization GUI for the Mobile Robot Using the A* Algorithm and Dijkstra Algorithm

For the following figures, we used the A* algorithm for mobile robot navigation.

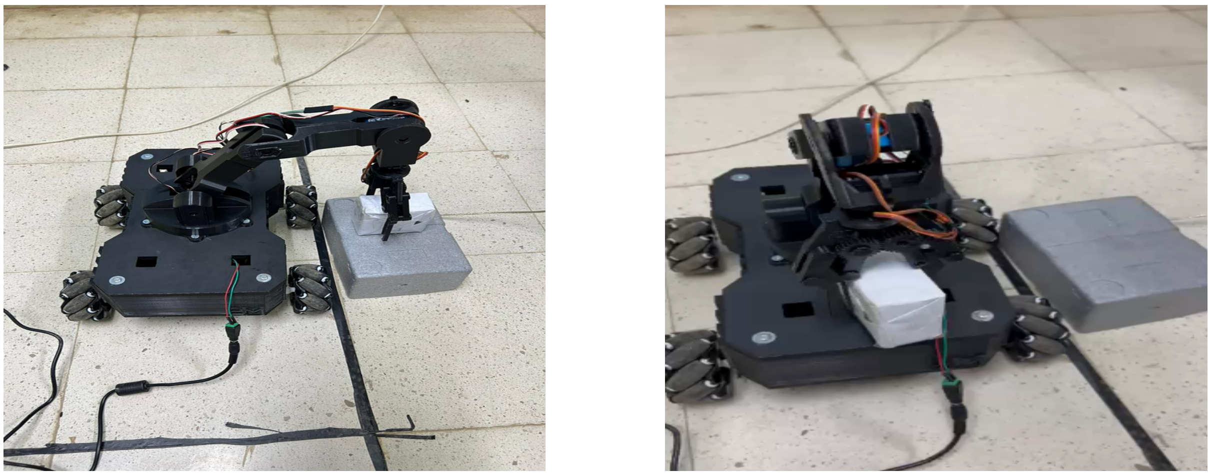


Figure 5.25: Start Position of the Mobile Robot and Robotic Arm

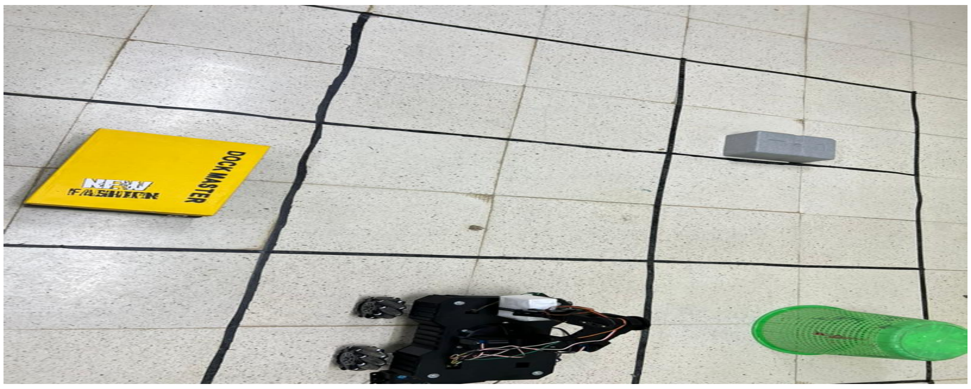


Figure 5.26: Position (0, 1) for the Mobile Robot and Robotic Arm

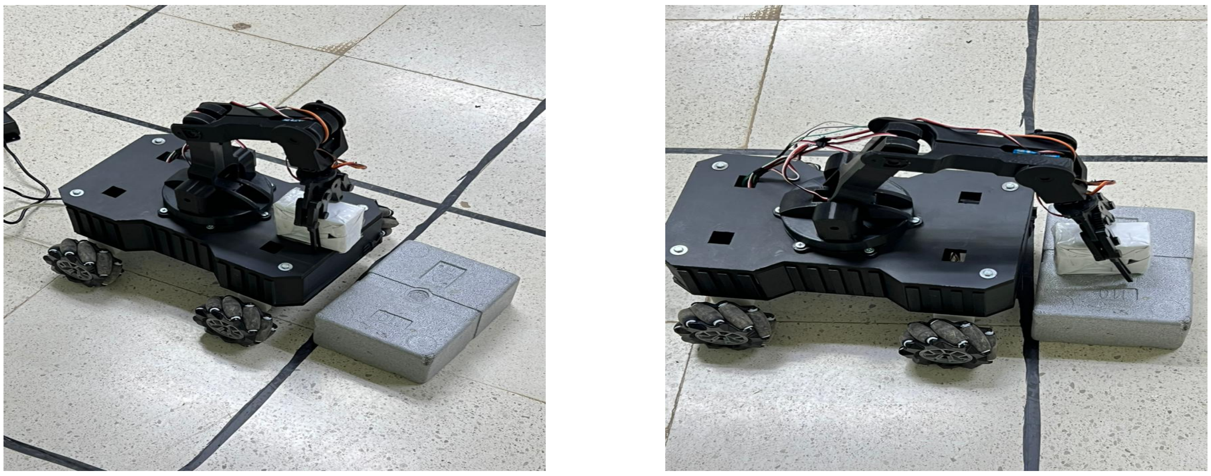
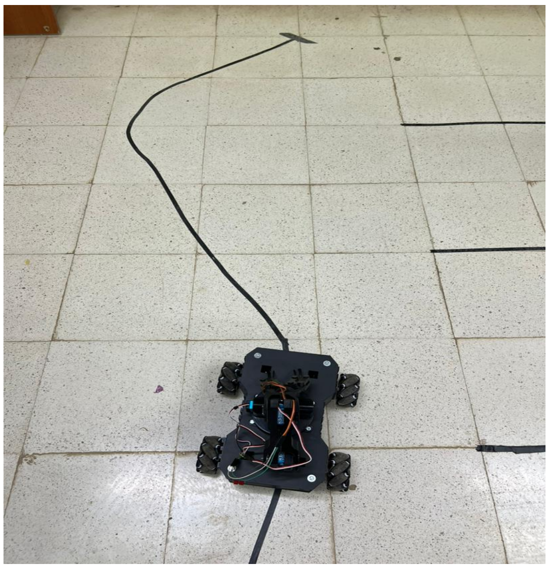


Figure 5.27: Goal Position of the Mobile Robot and Robotic Arm

5.7.4 Line-Following Robot GUI

The GUI for the line-following robot effectively displays the robot's operational status through a series of LED indicators. These indicators, which represent states such as *STOP*, *SHARP_LEFT*, *LEFT*, *FORWARD*, *RIGHT*, and *SHARP_RIGHT*, are color-coded to provide immediate visual feedback on the robot's behavior.
In operation, the GUI continuously updates the LED colors based on real-time data received via serial communication. Each status update triggers a change in the corresponding LED indicator's color, reflecting the robot's current activity or state.
For example, when the robot is moving forward, as shown in Figure (5.28), the *FORWARD* LED illuminates green, while other statuses are represented in red when inactive. This dynamic display allows for straightforward monitoring and assessment of the robot's performance.



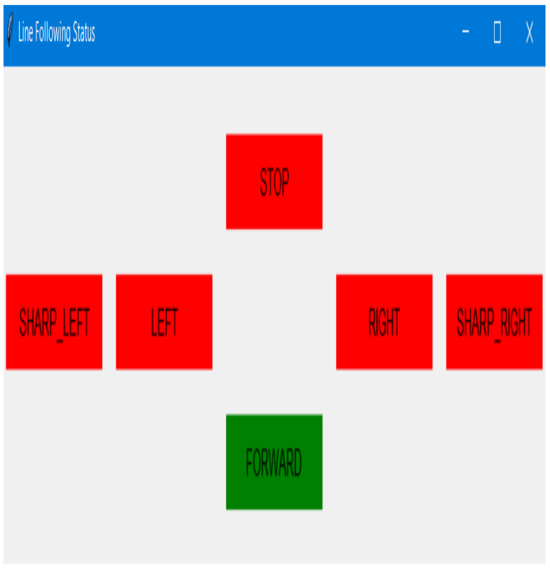


Figure 5.28: Robot Moving Forward While Following a Line

Similarly, Similarly, when the robot turns right, as shown in Figure (5.29), the *RIGHT* LED is highlighted, providing clear visual feedback on the robot's directional change.

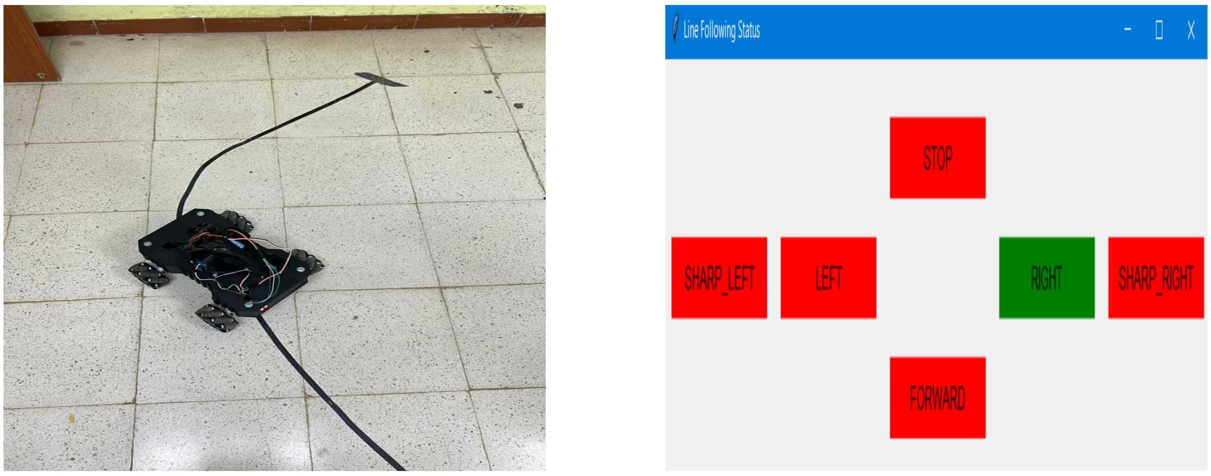


Figure 5.29: Robot Turning Right While Following the Line

5.7.5 Real-Time QR Code Detection System GUI

The system comprising a `Raspberry Pi 4`, `Raspberry Pi Camera Module v2`, and a gimbal with a servo motor is designed for high-precision, stabilized image and video capture. The `Raspberry Pi 4` acts as the central processor, handling both the control of the servo motor and the processing of visual data from the camera. The Camera Module v2 provides high-resolution imaging for applications such as object tracking and video streaming. The gimbal, driven by the servo motor, stabilizes the camera and allows it to rotate. This setup is suitable for various uses, including robotics, surveillance, and real-time computer vision, offering dynamic camera control and stable, high-quality imaging.

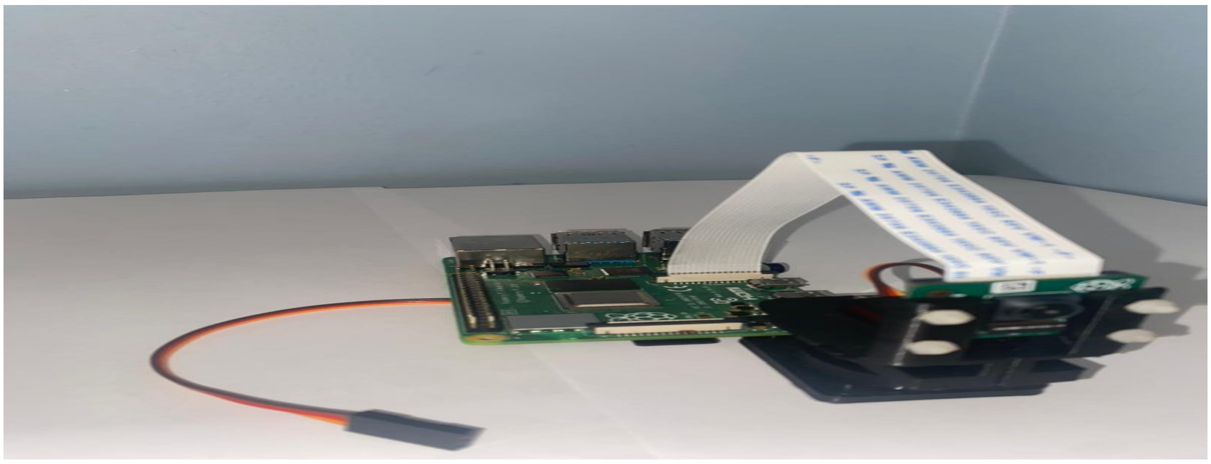


Figure 5.30: Assembly of Raspberry Pi 4, Camera Module v2, Gimbal, and Servo Motor Components

The graphical user interface (GUI) for the real-time QR code detection system integrates a live camera feed from a Raspberry Pi camera module. The system's primary feature is its ability to detect QR codes in real time, highlighting them with bounding boxes on

the live feed, as shown in Figure (5.31).
In addition to the visual representation, the GUI displays the decoded information from the QR code in a dedicated text field. The system also provides the center coordinates of the QR code in pixel terms (u, v), offering precise location details.
The interface includes controls for starting and stopping the camera feed, adjusting camera settings, and saving detected QR codes for further analysis. The design is user-friendly, ensuring intuitive navigation and efficient real-time processing of QR codes.

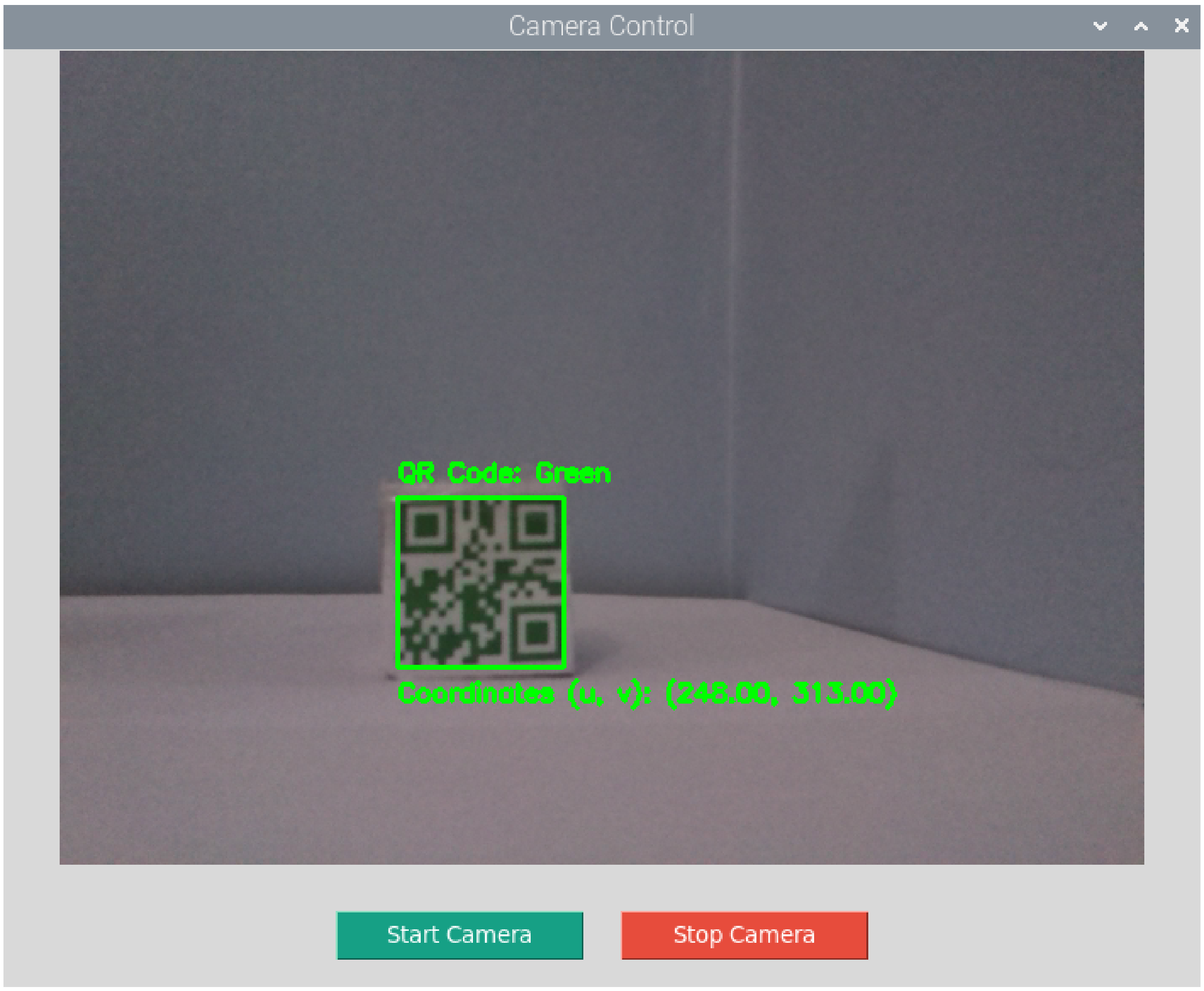


Figure 5.31: Real-Time QR Code Detection with Center Coordinates

The system also provides the distance between the camera and the QR code, measured in centimeters. This distance measurement is crucial for assessing the accuracy of the QR code detection and ensuring proper alignment during processing. The distance is displayed in the GUI, as illustrated in Figure (5.32), allowing users to monitor and adjust their setup as needed.

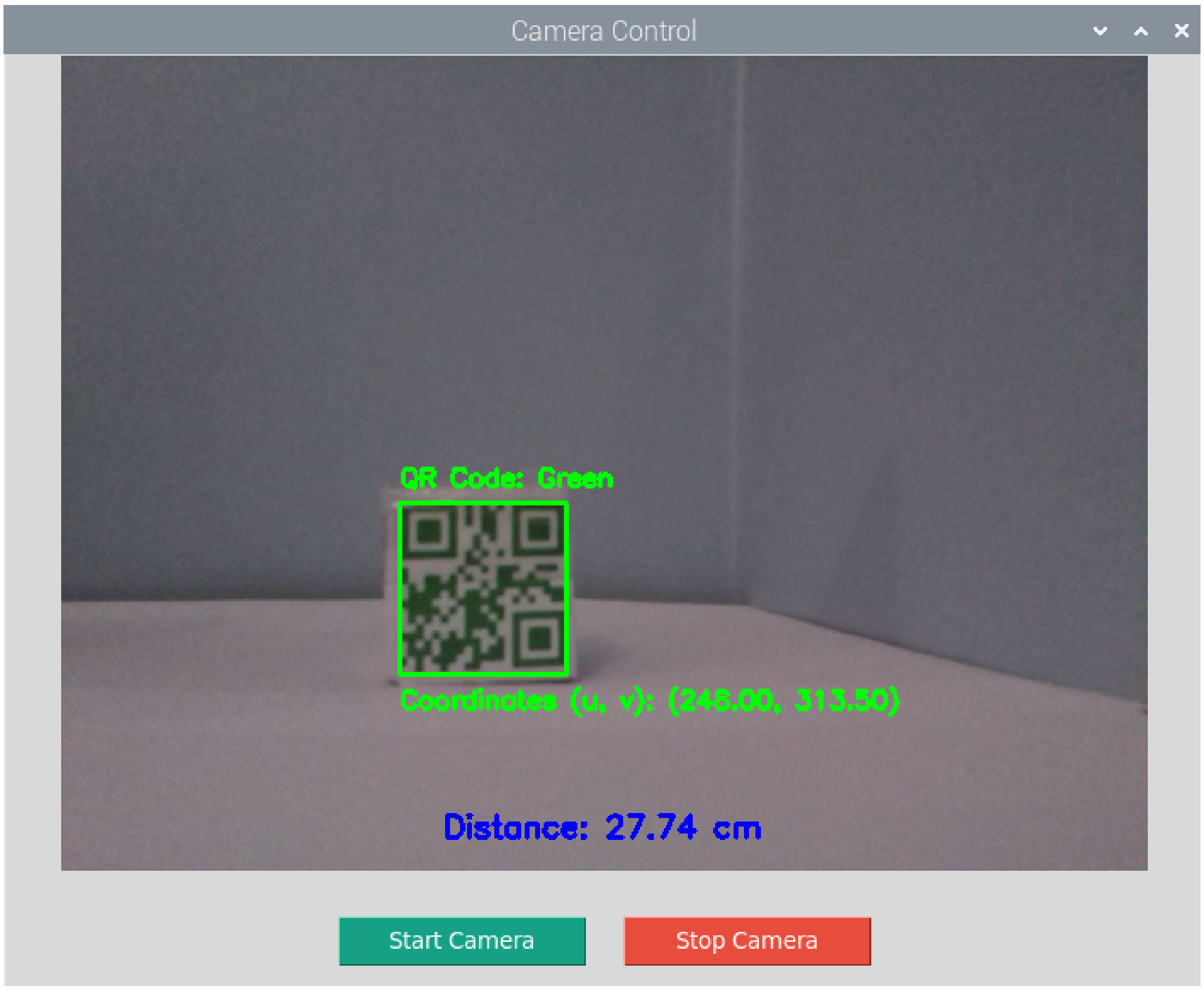
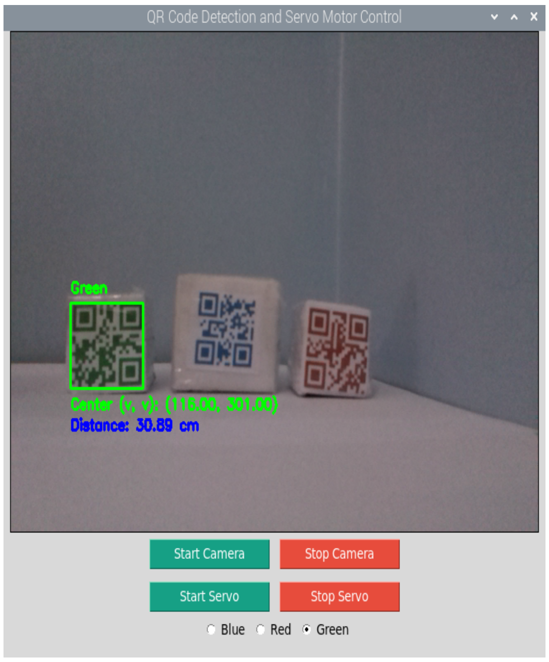


Figure 5.32: Distance Measurement Between the Camera and the QR Code

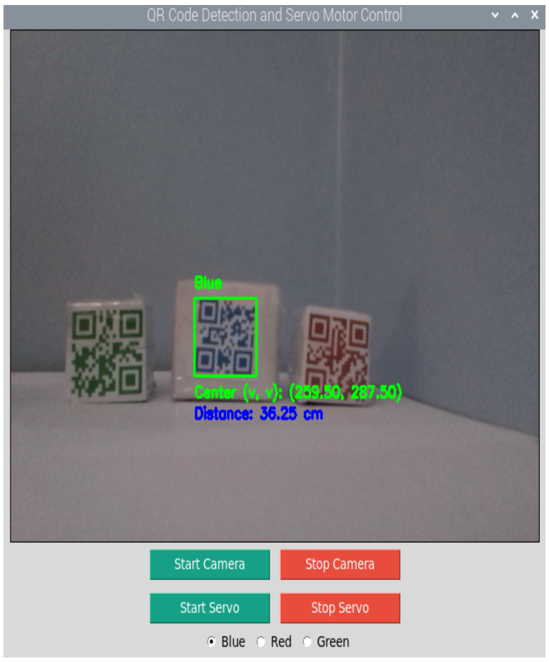
5.7.6 Real-Time QR Code Detection and Servo Motor Control GUI

The GUI for real-time QR code detection and servo motor control integrates video analysis with mechanical actuation. It continuously processes the live video feed to identify QR codes, highlighting those that match user-selected categories such as `"Blue"`, `"Red"`, or `"Green"` with a bounding rectangle and overlaying relevant information on the feed. The GUI also controls a servo motor, which is activated by pressing the `"Start Servo"` button. The motor's operation is closely linked to the QR code detection process, it rotates until the specified QR code is detected, at which point the system automatically stops the motor. This ensures a coordinated response between the visual detection of QR codes and the servo motor's mechanical actions, as is illustrated in Figure (5.33).



```
Center (v, v): (116.00, 301.00)
Distance: 30.89 cm
```

Blue [✓ Green]



```
Center (v, v): (259.50, 287.50)
Distance: 36.25 cm
```

[✓ Blue] Red Green

Green

Blue

Figure 5.33: Real-Time QR Code Detection with Servo Motor Control Interface

General Conclusion

This project report presents a comprehensive exploration of robotics systems and components, emphasizing the integration of theoretical foundations with practical applications. The study begins with a thorough examination of the fundamental principles underlying robotic control, including control theory, command architectures, and sensor integration. The theoretical foundation sets the stage for understanding various control strategies, which are further explored in the context of mobile robots, manipulator arms, and mobile manipulator robots.

The modeling section delves into the kinematic and dynamic aspects of robotic systems, providing detailed insights into Mecanum wheel models, kinematic constraints, and trajectory generation. By employing both forward and inverse kinematics, the report lays a solid foundation for understanding the motion planning and control of robotic arms.

In the trajectory generation and planning section, the report covers key algorithms such as Dijkstras and A* algorithms, offering a historical perspective, detailed descriptions, pseudocode, and comparative analysis. These algorithms are essential for pathfinding and navigation tasks, ensuring efficient and accurate movement of robotic systems.

Simulation plays a crucial role in validating the theoretical models and algorithms. The report includes simulations of robotic manipulator arms and mobile robots using MATLAB and GAZEBO, demonstrating how these tools can be leveraged to test and refine robotic systems.

Implementation aspects focus on practical considerations, including the use of Raspberry Pi, Arduino, and various communication protocols. The integration of OpenCV for computer vision applications and the development of graphical user interfaces (GUIs) using Tkinter are highlighted. These GUIs facilitate real-time control, pathfinding visualization, and QR code detection, underscoring the importance of user-friendly interfaces in robotics.

Overall, this project underscores the synergy between theoretical knowledge and practical implementation, providing valuable insights into the design, simulation, and control of advanced robotic systems.

Bibliography

[1] B. Siciliano and L. Khatib. *Robotics: Modelling, Planning and Control.* Springer, Berlin, 2010.

[2] E. Dickmanns and S. D. R. Schmidt. Real-time control systems in robotics. *International Journal of Robotics Research*, 26(4):373–389, 2007.

[3] E. Camacho and C. Bordons. Model predictive control. In *Advanced Control Engineering*, pages 45–68. Springer, 2013.

[4] H. R. Everett. *Sensors for Mobile Robots: Theory and Applications.* Aerospace Press, New York, 2006.

[5] R. E. Kalman. A new approach to linear filtering and prediction problems. In *Transactions of the ASME Journal of Basic Engineering*, volume 82, pages 35–45, 1960.

[6] M. Isard and A. Blake. Condensationconditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.

[7] J. J. Craig. *Introduction to Robotics: Mechanics and Control.* Prentice Hall, New Jersey, 2001.

[8] Igor Zeidis and Klaus Zimmermann. Dynamics of a four-wheeled mobile robot with mecanum wheels, 2019.

[9] J. Denavit and R. S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of Applied Mechanics*, 22(2):215–221, 1955.

[10] J. J. Craig. *Introduction to Robotics: Mechanics and Control.* Pearson, New Jersey, 3rd edition edition, 2005.

[11] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[12] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[13] Nils J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, 1984.

[14] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press, 2005.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020.

[16] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[17] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer, Berlin, Germany, 2nd edition, 2017.

[18] Inc. The MathWorks. *Simscape Multibody User's Guide*, 2023. Available: https://www.mathworks.com/help/physmod/sm/.

[19] Articulated Robotics. Describing robots with urdf, 2023. Accessed: 2024-09-14.

[20] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, 2016.

[21] Python Software Foundation. *Tkinter Documentation*, 2023. Available at https://docs.python.org/3/library/tkinter.html.

[22] Xin Zhao and Zhiqiang Zhang. Real-time qr code detection and decoding using computer vision. *Journal of Computer Vision and Image Understanding*, 172:1–11, 2018.