CYPRUS INTERNATIONAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

INDUSTRIAL AUTONOMOUS GUIDED VEHICLE

By

HASHEM VASEGHI

PARFAIT ZAINA NGOI

FELLY NGOY

JUDE KABEMBA

BOIMA FAHNBULLEH

GREGORY MWEMA

OLUTOYE OPEYEMI

March 24, 2025

Nicosia, NORTH CYPRUS

CYPRUS INTERNATIONAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

INDUSTRIAL AUTONOMOUS GUIDED VEHICLE

By

HASHEM VASEGHI

PARFAIT ZAINA NGOI

FELLY NGOY

JUDE KABEMBA

BOIMA FAHNBULLEH

GREGORY MWEMA

OLUTOYE OPEYEMI

March 24, 2025

Nicosia, NORTH CYPRUS

INDUSTRIAL AUTONOMOUS GUIDED VEHICLE

By

Hashem Vaseghi	22004087	Electrical and Electronic Engineering
Parfait Zaina Ngoi	22015208	Electrical and Electronic Engineering
Felly Ngoy	22013357	Computer Engineering
Jude Kabemba	22013160	Computer Engineering
Boima Fahnbulleh	22013081	Mechatronics Engineering
Gregory Mwema	22012501	Mechatronics Engineering
Olutoye Opeyemi	22013786	Mechanical Engineering

DATE OF APPROVAL: .....

APPROVED BY:

ASST. PROF. DR. ZİYA DEREBOYLU

Asst. Prof. Dr. ALİ SHEFIK

ACKNOWLEDGEMENTS

We extend our heartfelt appreciation to everyone who contributed to the success of this project. First and foremost, we express our deepest gratitude to **Asst. Prof. Dr. ZİYA DEREBOYLU** and **Asst. Prof. Dr. ALİ SHEFIK** for their invaluable guidance, continuous support, and encouragement throughout the project. Their expertise and insights were instrumental in overcoming challenges and ensuring the project's progress.

We are also grateful to **Cyprus International University** for providing us with the resources and platform to pursue this project. Special thanks to the faculty members of the **Faculty of Engineering** for their technical advice and mentorship, which greatly enriched our learning experience.

Our sincere thanks go to our entire team for their dedication and collaboration. Each member brought unique skills and perspectives, contributing to the successful integration of mechanical, electronic, and software systems. This project would not have been possible without their united commitment and hard work.

We also extend our gratitude to our families and friends for their unwavering support, encouragement, and motivation during the challenging phases of the project.

Finally, we would like to thank the organizers of the Teknofest Aerospace and Technology Festival for providing us with the opportunity to showcase our work and compete in the Digital Technologies in Industry competition.

This project is a testament to the collective effort and support of everyone involved. Thank you all for being part of this journey.

GITHUB REPOSITORY

you can access the project repository by scanning the QR code below or by clicking on the link below. you will find the pdf files of the project as well as the LaTeX file and the source code of the project.

Scan the QR code to access the project repository

ABSTRACT

The research introduces CIU_Fox as an autonomous guided vehicle (AGV) that develops capabilities to transform factory and warehouse internal transportation operations. The designed AGV features autonomous navigation with a safe lifting capability using obstacle detection and collision avoidance systems for moving loads up to 200 kg. This system combines LIDAR with time-of-flight (ToF) sensors and barcode scanners to create a precise automated navigation system which monitors operations and handles loading duties. The mechanical structure incorporates an aluminum alloy frame together with a scissor-compartment mechanism and NEMA 23 stepper-motor powered differential steering. A Raspberry Pi 4 manages the system through ESP32 modules that run software programs built in Python and C++ within the ROS framework. The robot development followed four sequential stages that led to its physical assembly. Resources limitations required the project team to conduct final stage testing through the Gazebo simulation pipeline instead of building physical components. Coding for path planning alongside obstacle avoidance and load management functions while achieving complete integration of mechanical electronic software system components stands as major accomplishments. The commercial application scope of the AGV remains promising in multiple industrial sectors including manufacturing storage facilities and logistics operations because it shows potential to optimize operational efficiency while decreasing expenses and protecting worker safety. This work illustrates why interdisciplinary teams need to bring innovative solutions to modern industrial design using state-of-the-art technology applications. Research efforts will focus simultaneously on two fronts which include enhancing the AGV's operational excellence while evaluating opportunities to connect it with broader industrial system networks.

ÖZET

Araştırma, fabrika ve depo içi taşıma operasyonlarını dönüştürme yetenekleri geliştiren özerk bir yönlendirmeli araç (AGV) olarak CIU_Fox'u tanıtmaktadır. Tasarlanan AGV, 200 kg'a kadar yük taşıyabilmek için engel algılama ve çarpışma önleme sistemleriyle donatılmış güvenli kaldırma kabiliyetine sahip otonom navigasyon özellikleri sunar.

Bu sistem, operasyonları izleyen ve yükleme görevlerini gerçekleştiren hassas bir otomatik navigasyon sistemi oluşturmak için LIDAR, zaman-uçuşu (ToF) sensörleri ve barkod tarayıcıları birleştirir. Mekanik yapı, alaşımlı alüminyum çerçeve, makaslı kompartman mekanizması ve NEMA 23 step motorla çalışan diferansiyel yönlendirme sistemi içerir.

Sistem, ROS çerçevesinde Python ve C++ ile geliştirilen yazılım programlarını çalıştıran ESP32 modülleri aracılığıyla bir Raspberry Pi 4 tarafından yönetilir. Robot geliştirme süreci, fiziksel montaja kadar uzanan dört aşamalı bir sırayla tamamlanmıştır. Kaynak kısıtlamaları nedeniyle fiziksel bileşenler yerine Gazebo simülasyon ortamında son aşama testleri gerçekleştirilmiştir. Yol planlama, engel önleme ve yük yönetimi fonksiyonlarının kodlanması ile mekanik-elektronik-yazılım bileşenlerinin tam entegrasyonu projenin temel başarıları arasındadır.

AGV'nin ticari uygulama kapsamı, üretim tesisleri, depolama alanları ve lojistik operasyonlar gibi çeşitli endüstriyel sektörlerde operasyonel verimliliği artırma, maliyetleri düşürme ve işçi güvenliğini koruma potansiyeli nedeniyle umut vaat etmektedir. Bu çalışma, disiplinlerarası ekiplerin en son teknoloji uygulamalarını kullanarak endüstriyel tasarıma yenilikçi çözümler getirmesi gerektiğini vurgulamaktadır. Araştırma çabaları, AGV'nin operasyonel mükemmelliğini artırmanın yanı sıra endüstriyel sistem ağlarına bağlanma fırsatlarını değerlendirmek üzere iki paralı hedefe odaklanacaktır.

Contents

List of Figures

List of Tables

List of Code Blocks

CHAPTER ONE
INTRODUCTION

The manufacturing industry is undergoing a digital transformation, driven by advancements in technologies like artificial intelligence, autonomous robots, and the Internet of Things. These innovations aim to enhance efficiency, productivity, and competitiveness in industrial processes. TEKNOFEST's Digital Technologies in Industry Competition challenges participants to design an autonomous guided robot for factory logistics, addressing real-world tasks such as navigation, load handling, and mapping. This report explores the strategies used to overcome these challenges and achieve the competition's objectives. By doing so, it highlights the critical role of digitalization in advancing industrial automation and fostering innovation.

1.1 TEKNOFEST COMPETITION RULES

The following rule apply to the operation of guided robots during the competition:

1.1.1 Autonomous Operation

The guided robot must perform all tasks autonomously within the specified scenarios, including line-following.

1.1.2 Overload Warning

The robot must issue an overload warning if it lifts a load exceeding the specified limit and continue carrying the load only after the weight is reduced below the limit.

1.1.3 Charging Task

If the robot's charge level falls below a certain threshold, teams can earn additional points by having the robot autonomously navigate to the charging area and begin charging. Teams must inform the jury in advance if they wish to perform this task, which will be conducted at a time deemed appropriate by the jury.

1.1.4 Obstacle Navigation

The robot should autonomously stop at loading/unloading points and navigate around obstacles if they are not removed. Obstacles will be detected by sensors, and the robot must stop at an appropriate distance. If the obstacle remains, the robot should autonomously go around it and complete its tasks.

1.1.5 QRCODE Labels

QRCODE labels at loading and unloading positions must be readable by appropriate sensors on the robot.

1.1.6 Control Screen (GUI)

Teams must prepare a graphical user interface (GUI) that allows them to monitor the vehicle's status and issue commands when necessary. However, interventions via the control panel will incur penalty points.

1.1.7 No Manual Control

The robot cannot be controlled by joysticks, portable hand controls, phones, or tablets.

1.1.8 Load Handling Display

The pick-up or dropping of loads must be displayed on the control panel (GUI) using sensors.

1.1.9 Mapping for Advanced Competitors

Advanced-level competitors are required to map the track. For mapping, advanced teams will be given a specific amount of time after the loaded course is revealed. The teams are expected to map and display the competition area within the allocated time.

Mapping should be performed using devices such as laptops and tablets belonging to the team members at the control desk. Additionally, teams are expected to create a control

panel (GUI) that displays competition details such as speed and total task time. The mapping should be detailed, and QRCODE labels should be displayed on the map as they are read.

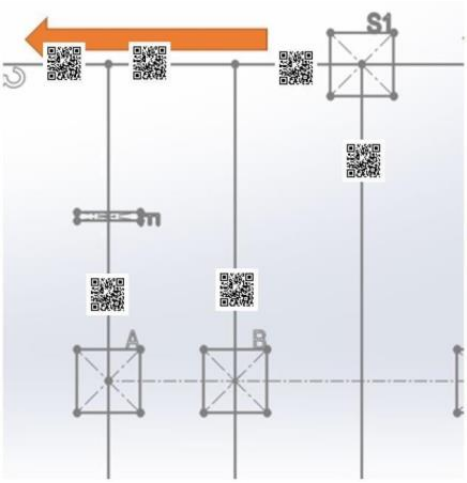


Figure 1.1: QRCODE Layout

1.2 DETAILS OF THE COMPETITION AREA

For the competition, there will be 2 rectangular-shaped representative factory areas of approximately 150 square metres each. These areas will include a track representing the layout of roads and internal logistics roads. Additionally, there will be a separate area where a table is located for each participating competitor team to use. The following details apply:

1.2.1 Power Supply

The competition area will have a 220 VAC power supply. A control desk will be located at the edge of the track for the competing team to control their guided robot. At the control desk, 220 VAC voltage will be provided, and each team is responsible for performing AC/DC conversion using their own converter. The highest DC voltage level that can be used is 50V.

1.2.2 Track Layout

The track will resemble a factory area with designated pick-up and drop-off points. The distribution of these points may vary for each competing team, as determined by the referees. The track area will consist of two similar sections, allowing two different teams to compete simultaneously.

1.2.3 Sample View of the Track

The sample view of the track will be provided separately for reference(fig. 1.2).

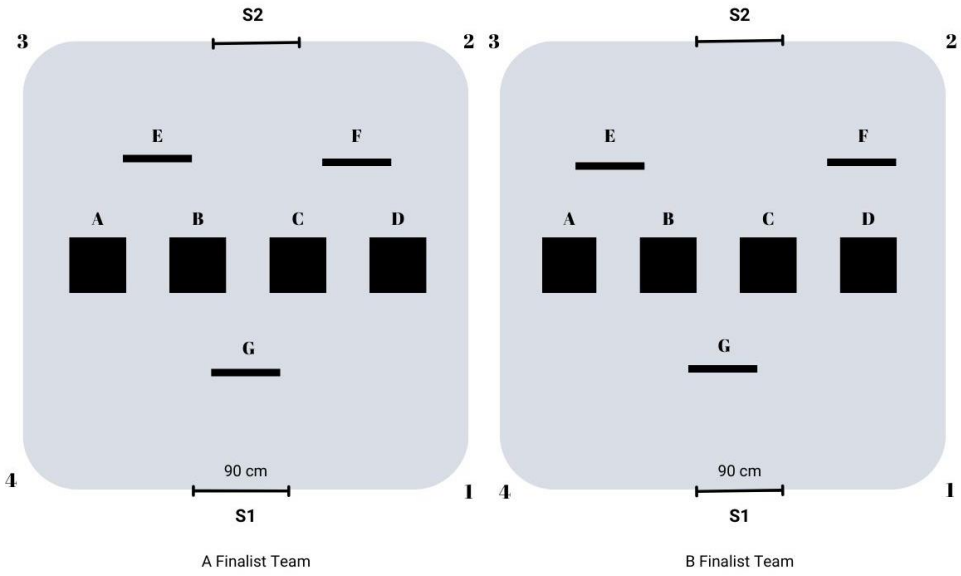


Figure 1.2: Sample Track Area

1.3 LOAD HANDLING ROBOT TECHNICAL SPECIFICATIONS AND LIMITATIONS

The maximum dimensions of the load handling robot are as follows: 1,000 mm in length, 900 mm in width, and 500 mm in height. The dimensions of the load handling robots must not exceed these specified limits. However, when the mechanism used to lift the load platform is operated, the robot may exceed the height limit. The height restriction applies only to the situation where the vehicle is not operating under load.

The maximum load amount that robots must carry is 125 kg. Robots that lift more than 125 kg and do not provide an overload warning will be deemed to have failed to fulfill this task. The competition organization will provide loads with dimensions of 30 cm x 30 cm x 10 cm (fig. 1.1a)and a weight of 25 kg each. These loads will be placed on a platform with a maximum height of 50 cm from the ground)(fig. 1.3b), allowing the robot to easily position itself underneath and lift the load from all directions. The weight of the load platform provided by the competition organization will also be 25 kg.

The positions for load pick-up and drop-off will be defined using QRCODE tags. These tags will ensure precise identification of the designated locations for the robot to perform its tasks.

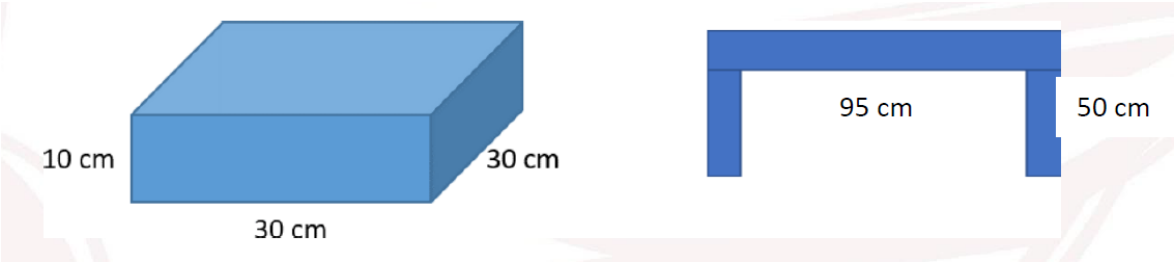


Figure 1.3: Load (a) and platform (b)

1.4 SAMPLE SCENARIO

The robot must first navigate around the corner points according to the scenario type, starting from the initial position without any load, and return to the starting point. During this process, teams are required to complete the mapping task. A total of four unloaded scenarios are illustrated in fig. 1.4.

Following the unloaded navigation, the robot should proceed to point A from the starting point to pick up a load and then continue to point C. Upon reaching point C, the robot must leave the load there and proceed to point D without carrying any load. After arriving at point D, the robot should pick up another load and transport it to point B. Once the load is delivered at point B, the robot must complete the task by returning to the starting point without any load. fig. 1.5 depicts four loaded scenarios, which vary depending on the starting points.

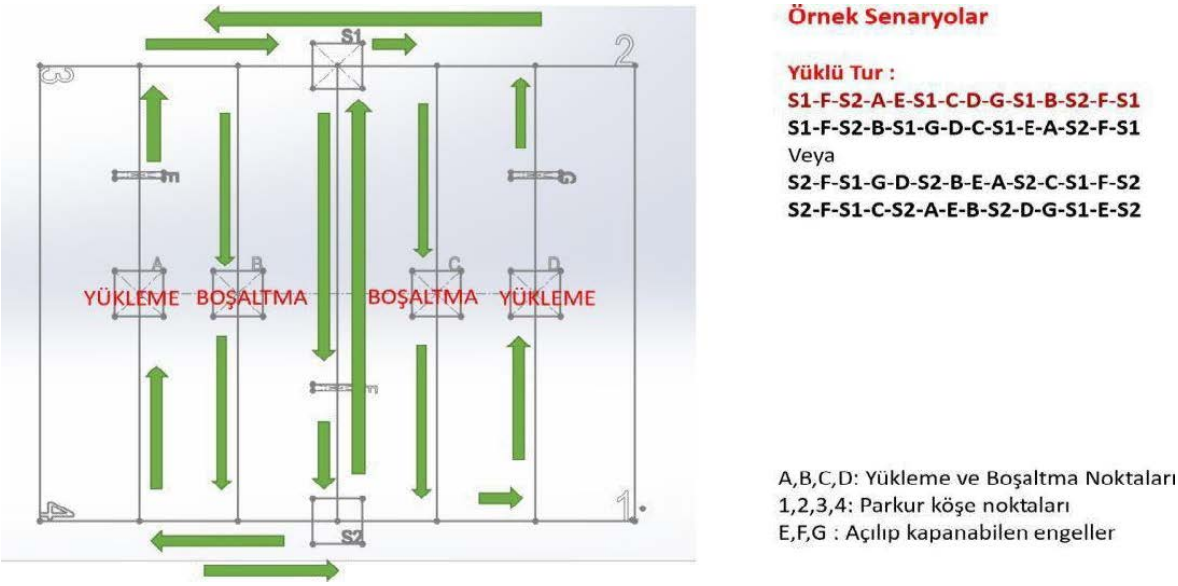


Figure 1.4: Sample Loaded Scenario

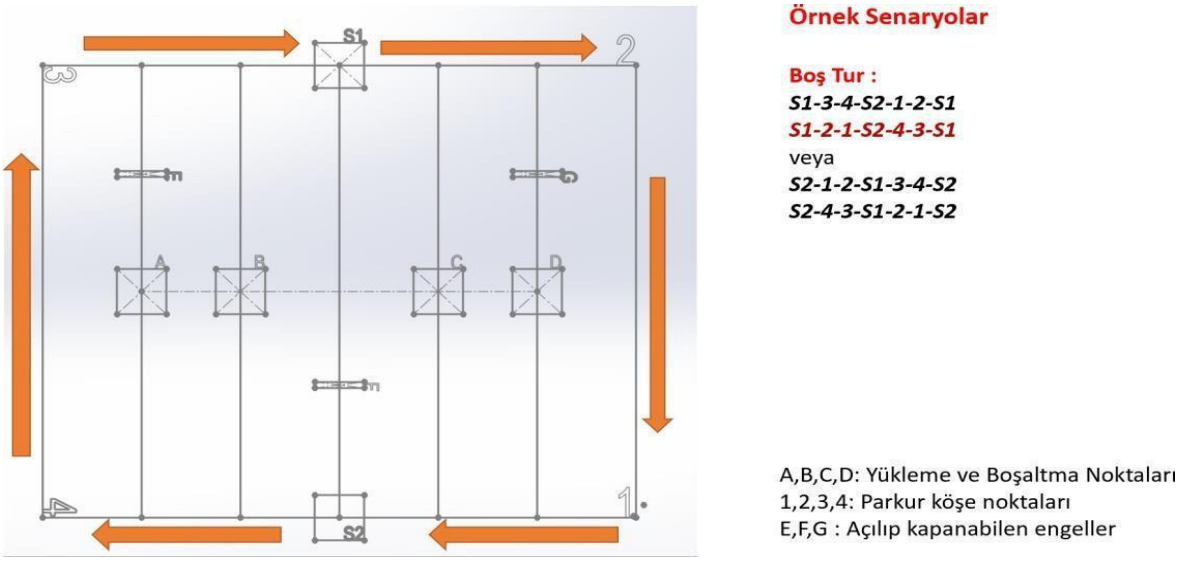


Figure 1.5: Example No Load Scenario

CHAPTER TWO
LITERATURE SURVEY

Today's industries activity have merged with the robotic and automation world and day by day having the precise and betterquality product make the sense of using new technology. Between all types of robot, the AGV (automated guided vehicle) robot has the special place between others and improvement in technology helps this design to grow and become more helpful in various applications. The AGV robot is a programmable mobile robot integrated sensor device that can automatically perceive and move along the planned path [2] This system consists of various parts like guidance facilities, central control system, charge system and communication system [3].

The initial used and Invention AGV is not clear exactly and was mentioned in different articles and reference for many times but the earliest time of using this system in industries is mentioned in 1950s [4] (fig. 2.1) and even mentioned in some reference that the first AGV in the world was introduced in UK in 1953 for transporting which was modified from a towing tractor and can be guided by an overhead wire [3].

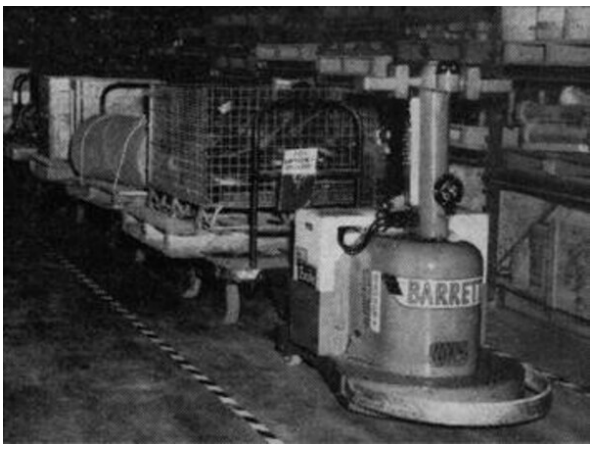


Figure 2.1: one of the Old AGV picture

AGVs are widely applied in various kinds of industries including manufacturing factories and repositories for materialhandling. After decades of development, it has a wide application due to its high efficiency, flexibility, reliability, safety and system scalability in various task and missions. AGV operates all day long continuously that cannot be achieved by hu-

man workers. Therefore, the efficiency of material handling can be boosted by having the collaborating task with number of AGV . In this case, administrator can enable more AGVs as the system is extensible. AGV has capability of collision avoidance and emergency braking, and generally the running status is monitored by control system so that reliability and safety are ensured. Generally, a group of AGVs are monitored and scheduled by a central control system. AGVs, ground navigation system, charge system, safety system, communication system and console make up an AGV system. [5]. This report examines the design aspects and considerations for this type of robot, providing readers with key insights into the technologies commonly utilized in this field.

2.1 KEY TECHNOLOGIES IN AGV DEVELOPMENT

2.1.1 Navigation Systems

Modern AGVs employ various navigation techniques, including:

- **Laser-guided navigation**: Uses LiDAR sensors to detect reflectors and map the environment.
- **Vision-based navigation**: Utilizes cameras and computer vision algorithms for path planning and obstacle avoidance.
- **Inertial navigation**: Relies on gyroscopes and accelerometers for position tracking.
- **Natural feature navigation**: Uses environmental features (e.g., walls, racks) for localization without predefined markers.

2.1.2 Localization and Mapping

Simultaneous Localization and Mapping (SLAM) algorithms are widely used for real-time environment mapping and AGV positioning. SLAM integrates data from sensors like LiDAR, cameras, and ultrasonic sensors to create accurate maps.

2.1.3 Path Planning and Control

AGVs use algorithms such as A*, Dijkstra's, or Rapidly-exploring Random Trees (RRT) for optimal path planning. Advanced control systems ensure smooth motion and collision avoidance.

2.1.4 Communication and Coordination

Multi-AGV systems rely on wireless communication protocols (e.g., Wi-Fi, 5G) and centralized or decentralized control strategies to coordinate tasks and avoid conflicts.

2.2 APPLICATIONS OF AGVS

AGVs are widely used in various industries:

- **Manufacturing**: For transporting raw materials, components, and finished products.
- **Warehousing**: For order picking, inventory management, and goods transportation.
- **Healthcare**: For delivering medical supplies and meals in hospitals.
- **Agriculture**: For automated harvesting and crop monitoring.

2.3 CHALLENGES AND LIMITATIONS

Despite their advantages, AGVs face several challenges:

- **High Initial Costs**: The development and deployment of AGVs require significant investment in hardware and software.
- **Dynamic Environments**: AGVs struggle in unstructured or highly dynamic environments with moving obstacles.
- **Battery Life and Charging**: Limited battery capacity and the need for frequent charging can disrupt operations.
- **Safety Concerns**: Ensuring safe interaction with human workers and other equipment is critical.

2.4 RECENT TRENDS AND INNOVATIONS

- **Integration with AI and Machine Learning**: AGVs are increasingly leveraging AI for predictive maintenance, adaptive navigation, and task optimization.
- **Collaborative AGVs**: Development of AGVs that can work alongside humans (cobots) is gaining traction.
- **Swarm Robotics**: Research is exploring the use of multiple AGVs working collaboratively in a decentralized manner.

- **Autonomous Mobile Robots (AMRs)**: AGVs are evolving into AMRs with higher autonomy and flexibility.

2.5 NOTABLE RESEARCH AND CASE STUDIES

- **Research on SLAM for AGVs**: Studies have focused on improving SLAM algorithms for better accuracy and robustness in dynamic environments.
- **Energy-Efficient AGVs**: Research has explored energy-saving techniques, such as regenerative braking and optimized path planning.
- **Human-AGV Interaction**: Studies have investigated intuitive interfaces and safety protocols for human-AGV collaboration.

CHAPTER THREE
REALISTIC CONSTRAINTS

3.1 DESIGN CONSTRAINTS

3.2 ENGINEERING STANDARDS AND LIFELONG LEARNING

The successful deployment of an Automated Guided Vehicle (AGV) system relies heavily on adherence to established engineering standards. These standards provide a framework for the design, implementation, and operation of AGVs, ensuring they meet safety, reliability, and performance requirements.

Compliance with these guidelines is essential not only for regulatory approval but also for fostering a culture of continuous learning and improvement in AGV technology and its applications.

3.2.1 AGV Safety Standards and Their Importance

Various international standards govern the safety of AGVs, ensuring their integration into industrial, healthcare, and logistics environments without compromising human safety.

Notable among these are **ISO 3691-4** [6], which outlines safety requirements for driverless industrial trucks, and **ANSI/ITSDF B56.5** [7], which provides American safety guidelines for automated guided industrial vehicles.

The **EN 3691-4** [8] standard mirrors the ISO regulations for European markets, emphasizing risk reduction and operational reliability. Meanwhile, **RIA R15.08** [9]is an evolving effort to create comprehensive safety guidelines for mobile robots in industrial settings.

These standards establish essential safety features for AGVs, including **emergency stop mechanisms, audible alarms, warning lights, and collision avoidance systems**.

They also define environmental considerations such as clear pathways, adequate lighting, and obstacle detection to enhance the safe operation of AGVs.

Additionally, operational protocols for **startup, shutdown, emergency response, and periodic maintenance** are emphasized to prevent malfunctions and ensure continuous performance.

3.2.2 Categorization of Machinery Safety Standards

Engineering safety standards are categorized into three types to ensure a structured approach to risk mitigation:

- **Type-A Standards:** These provide general safety principles applicable to all machinery, focusing on fundamental design requirements.

- **Type-B Standards:** These cover protective devices and safety measures that apply to a wide range of machines, ensuring standardization across different industries.

- **Type-C Standards:** These are specific to particular types of machinery, including AGVs, and take precedence over Type-A and Type-B standards when addressing specific risks.

While these standards ensure a structured approach to AGV safety, they do not account for additional hazards such as **severe environmental conditions, nuclear operations, or public-road navigation**, necessitating customized engineering solutions for such applications.

3.2.3 Designing AGVs for Public Interaction

Most AGVs are deployed in controlled industrial environments, operated by trained professionals.

However, in certain sectors, they may interact with **untrained personnel, visitors, or even the general public**.

For instance, in healthcare settings, robots used for hospital logistics must safely coexist with **doctors, nurses, patients, and visitors** who may be unfamiliar with automated systems.

Similarly, in retail and hospitality sectors, AGVs designed for service roles must incorporate intuitive safety mechanisms and user-friendly interfaces to prevent accidents.

To ensure safe public interaction, they must be designed with **intelligent obstacle detection, adaptive navigation, and fail-safe mechanisms** that allow them to adjust their behavior in real time.

Features such as **voice alerts, digital displays, and intuitive stop/start functionalities** help bridge the gap between automation and human interaction.

3.2.4 Lifelong Learning and Continuous Development in AGV Engineering

The field of AGV technology is rapidly evolving, necessitating **continuous education and lifelong learning** among engineers, operators, and industry stakeholders.

Given the advancements in **artificial intelligence, sensor technologies, and machine learning**, professionals must stay updated on emerging safety protocols, regulatory changes,

and new engineering methodologies.

Training programs, certifications, and industry workshops play a crucial role in ensuring that **engineers, technicians, and operators** remain proficient in the latest AGV technologies.

Furthermore, the increasing adoption of **Robot-as-a-Service (RaaS)** models means that companies must not only invest in AGV technology but also continuously train their workforce to adapt to evolving automation trends.

Engineering standards form the backbone of AGV safety, reliability, and efficiency, guiding their deployment across various industries.

The integration of **rigorous safety measures, structured categorization of standards, and adaptive public interaction mechanisms** ensures that AGVs can function seamlessly while maintaining workplace safety.

Additionally, the fast-paced evolution of AGV technology demands a culture of **lifelong learning and professional development**, enabling engineers and industry professionals to keep pace with advancements in automation and robotics.

Through adherence to standards and continuous education, businesses can maximize the benefits of AGVs while fostering a safer, more efficient work environment

3.3 ECONOMIC ANALYSIS OF INDUSTRIAL AUTOMATED GUIDED VEHICLES (AGVS)

Developing Automated Guided Vehicles (AGVs) requires a substantial financial investment for engineers and companies aiming to automate material handling and logistics operations. While AGVs significantly enhance efficiency and productivity, a thorough economic analysis is essential to assess direct costs, such as equipment and installation, as well as indirect costs, including unforeseen expenses.

A well-planned budget and strategic approach ensure that the cost of building AGVs remains within the initial projected expenditure, preventing cost overruns and maximizing return on investment. Building Automated Guided Vehicles (AGVs) in the TRNC presents unique financial challenges due to the region's geographical location and limited local manufacturing capabilities. One of the most significant hurdles is the complexity of sourcing components, which often need to be imported from various countries such as China, Germany, and the United States. This reliance on international suppliers introduces several direct and indirect costs that can strain budgets and complicate project timelines.

3.3.1 Sourcing Components Across Borders

The construction of AGVs requires highly specialized parts, including advanced sensors, navigation systems, robotic arms, and durable materials for vehicle frames. Many of these components are not readily available locally and must be ordered from manufacturers abroad. For example, high-precision sensors may come from Germany, battery systems from the United States, and certain electronic modules from China.

The process of coordinating shipments from multiple countries adds layers of complexity, delays, and additional expenses. Furthermore, some parts are custom-made to meet the specific requirements of the Teknofest competition, leading to longer lead times and higher procurement costs.

3.3.2 Direct Costs: Purchasing Parts and Shipping

The direct costs associated with building AGVs in the TRNC include the purchase price of components and shipping fees. Importing parts from distant countries like China or the U.S. involves significant freight charges, especially for bulky or heavy items such as motors and chassis materials.

Additionally, currency exchange rates can further inflate costs, as fluctuations may increase the price of goods purchased in foreign currencies. These factors make it difficult to accurately forecast expenditures and maintain budgetary control during the development phase.

3.3.3 Indirect Costs: Taxes, Customs Fees, and Regulatory Compliance

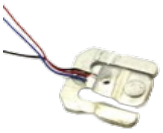
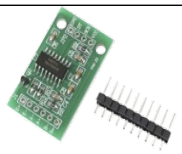
Beyond the direct costs, there are substantial indirect costs tied to importing parts into the TRNC. Customs duties, value-added taxes (VAT), and other regulatory fees can add a considerable percentage to the overall cost of components. For instance, certain high-tech equipment may attract steep import tariffs, while VAT rates in the region can further escalate expenses.

These regulations requires expertise and administrative effort, which can divert resources away from core engineering tasks. Moreover, delays at customs due to paperwork issues or inspections can disrupt production schedules, causing additional financial strain.

3.3.4 COST

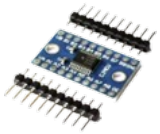
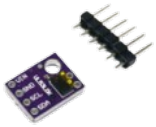
The following table provides a detailed breakdown of the estimated costs associated with building an AGV.

Table 3.1: List of Components and Their Costs

Components	Quantity	Price (per Unit)	Reference Image
Raspberry Pi 4 8GB RAM	1	3127.03 ₺	
ESP32-S3-DevKitC-1-N8R8 - ESP32-S3-WROOM-1	2	1220 ₺	
Closed Loop Stepper Driver V4.1 0-8.0A 24-48VDC CL57T	2	1186 ₺	
Motorobit Weight Sensor 120 kg	2	768.2 ₺	
Weight Sensor - Load Sensor 50Kg.	4	29.7 ₺	
Load Cell Amplifier - HX711	4	27.3 ₺	
BTS7960B 40 Amp Motor Driver Board	1	188.82 ₺	
Barcode Scanner Module 1D/2D Codes Reader	1	1261.83 ₺	
QTRXL-MD-01A Reflectance Sensor Array	4	90.57 ₺	

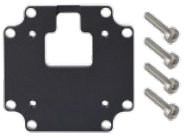
Continued on next page

Table 3.1 – continued from previous page

Components	Quantity	Price (per Unit)	Reference Image
Gravity: HUSKYLENS	1	1723.14 ₺	
IMU Sensor / 9 Axis MPU9255 IMU and Barometric Sensor (Low Power)	1	1058.31 ₺	
RPLIDAR - 360 degree Laser Scanner Development Kit	1	5029.72 ₺	
TXS0108E 8 Channel Voltage Level Transducer	4	40 ₺	
The VL53L0X is a time-of-flight (ToF) distance sensor	10	101.76 ₺	
UV Solder Mask	3	180.78 ₺	
LM2596HV/LM2576 Voltage Regulator for Multiple power supply	4	36.09 ₺	
Aluminum heatsink	2	439 ₺	
5V 8 Channel Relay Card	1	154.68 ₺	

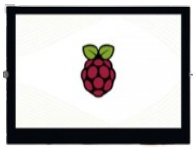
Continued on next page

Table 3.1 – continued from previous page

Components	Quantity	Price (per Unit)	Reference Image
P Series Nema 23 Closed Loop Stepper Motor 2Nm(283.28oz.in) with Electromagnetic Brake	2	2971.78 ₺	
EG Series Planetary Gearbox Gear Ratio 20:1 Backlash 20arc-min for 10mm Shaft Nema 23 Stepper Motor	2	1642 ₺	
Shaft Sleeve Adaptor 11mm to 8mm for NMRV30 Worm Gearbox	4	35 ₺	
Single Output Shaft for NMRV30 Worm Gearbox	4	121.37 ₺	
Double Output Shaft for NMRV30 Worm Gearbox	4	121.37 ₺	
NEMA 23 Stepper Motor Vibration Damper	4	243.74 ₺	
Nema 23 Bracket for Stepper Motor	4	243.74 ₺	
Nema 23 Flange for ISC And ISD Series Drivers	3	175.28 ₺	
AWG 20 High-flexible with Shield Layer Stepper Motor Cable	2	43.25 ₺	

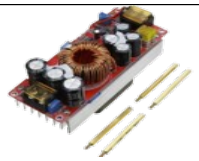
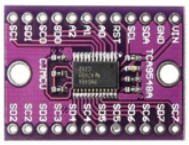
Continued on next page

Table 3.1 – continued from previous page

Components	Quantity	Price (per Unit)	Reference Image
TP-Link TL-WR840N	1	569 ₺	
Raspberry Pi 4.3 Inch Capacitive Touch Screen DSI Interface 800x480	1	1750.28 ₺	
Nema 8R 5W 87dB 90x39mm Speaker	1	108.11 ₺	
Nema RS232 to Bluetooth Series Adapter	2	455 ₺	
12V 70 AH AGM BATTERY	1	5000 ₺	
Battery Chargers 6V/2A 12V/2A Full Automatic Smart Battery	1	642.99 ₺	
RS232 Adapter Cable to USB 2.0	2	453.65 ₺	
Motor and encoder extension cable kit	2	351 ₺	
DC 12V Electric Linear Actuator Force 6000N	1	2479 ₺	

Continued on next page

Table 3.1 – continued from previous page

Components	Quantity	Price (per Unit)	Reference Image
WM-045 DC-DC 150W Voltage Booster	1	521.04 ₺	
Motorobit DC-DC 1500W 30A Voltage Boost Module	1	881.76 ₺	
TCA9548A I2C Multiplexer Card	1	40.66 ₺	
3B Printer Limit Switch	2	37.07 ₺	
Drn956 16mm Emergency Stop Switch (Head 27mm)	1	145.08 ₺	
		Total Price	**82945.29 ₺**

3.4 SUSTAINABILITY

The integration of Automated Guided Vehicles (AGVs) in material handling and logistics has ushered in a new era of sustainability, offering significant advantages over traditional methods such as forklifts.

As industries increasingly prioritize environmentally friendly solutions, these vehicles have emerged as a critical component in reducing environmental impact while simultaneously improving operational efficiency. Their ability to operate on electric power, optimize movement, and minimize waste positions them as a sustainable choice for modern warehouses and logistics operations.

One of the most notable contributions to sustainability is their energy efficiency. Unlike conventional forklifts, which rely on fossil fuels and emit harmful pollutants, these systems are electrically powered, producing zero direct emissions. Equipped with advanced route optimization and intelligent navigation, they ensure the most efficient paths, reducing unnecessary energy consumption.

Furthermore, many modern models utilize lithium-ion batteries, which offer longer operational cycles and shorter charging times compared to traditional lead-acid batteries. Some even incorporate regenerative braking technology, enabling them to recover and reuse energy that would otherwise be lost, further enhancing their efficiency.

Another key advantage lies in their ability to reduce waste. With precise movement capabilities and advanced sensor technology, these systems minimize product damage during transportation, significantly lowering material waste. Traditional equipment, such as forklifts, often leads to inventory losses due to human error or accidents.

In contrast, these automated systems are designed to handle goods with care and accuracy, preserving inventory integrity. By preventing unnecessary waste and reducing the need for product replacements, they contribute to a more sustainable and cost-effective supply chain. Beyond energy efficiency and waste reduction, these vehicles also support sustainability through optimized space utilization and reduced labor dependency. Their compact design and ability to operate in tight spaces allow warehouses to maximize storage capacity, reducing the need for expansive facilities.

Additionally, they can operate continuously without fatigue, minimizing reliance on human labor and lowering operational costs over time. Beyond energy efficiency and waste reduction, these vehicles also support sustainability through optimized space utilization and reduced labor dependency. Their compact design and ability to operate in tight spaces allow warehouses to maximize storage capacity, reducing the need for expansive facilities. Additionally, they can operate continuously without fatigue, minimizing reliance on human labor and lowering operational costs over time.

3.5 ETHICAL IMPLICATIONS OF AGVS IN LOGISTICS AND MATERIAL HANDLING

Automated Guided Vehicles (AGVs) in logistics and material handling brings not only sustainability benefits but also significant ethical challenges. While AGVs enhance efficiency and reduce environmental impact, their adoption raises critical concerns related to job displacement, data privacy, workplace safety, and environmental responsibility. Addressing these issues is essential for businesses to ensure ethical and responsible implementation.

3.5.1 Job Displacement and Workforce Transition

The widespread adoption of AGVs and autonomous systems has the potential to disrupt not only individual workers but also entire communities and economies. As these technologies replace human labor in industries such as logistics, warehousing, and transportation, the ripple effects extend beyond job loss. Local economies that depend on these industries may

experience reduced consumer spending, declining tax revenues, and increased demand for social services, creating a cycle of economic stagnation.

Furthermore, the displacement of workers in lower-wage, manual labor roles can lead to broader societal challenges, such as increased income inequality and reduced social mobility. Communities heavily reliant on these jobs may face higher rates of poverty and unemployment, exacerbating existing social divides. On a macroeconomic level, the shift toward automation could alter labor market dynamics, potentially leading to a mismatch between available jobs and the skills of the workforce.

3.5.2 Data Privacy and Security

Another ethical challenge is the handling of data collected by AGVs. These systems rely on advanced sensors, cameras, and AI-driven software to navigate and optimize operations. In the process, they gather vast amounts of data on operational efficiency, movement patterns, and even worker behavior. Without proper management, this data could be misused, leading to concerns about workplace surveillance and privacy violations.

3.5.3 Workplace Safety

Safety is a critical ethical consideration in AGV deployment. While AGVs are designed to reduce accidents and enhance workplace safety, their interaction with human workers requires careful oversight. Malfunctions, software errors, or unexpected obstacles could lead to accidents if safety protocols are not strictly followed. Employers must ensure that AGVs are equipped with reliable collision avoidance systems and that employees receive adequate training to work alongside automated systems. Regular maintenance and system updates are also essential to prevent operational failures that could endanger workers.

3.5.4 Environmental Responsibility

Beyond operational sustainability, the environmental impact of AGV production and disposal raises ethical concerns. While AGVs contribute to greener logistics during their operation, their manufacturing involves resource-intensive components such as lithium-ion batteries. Responsible sourcing of materials, fair labor practices in manufacturing, and proper recycling programs for outdated AGVs are essential to minimize their environmental footprint. Companies must prioritize ethical supply chain practices to ensure that AGVs align with broader sustainability goals.

3.5.5 Balancing Technology and Ethics

The ethical deployment of AGVs requires a careful balance between technological advancement and corporate responsibility. Businesses must proactively address concerns related to employment, data privacy, safety, and environmental impact to ensure that AGVs benefit both operations and society. By adopting ethical frameworks alongside technological innovations, companies can harness the advantages of AGVs while upholding fairness, transparency, and sustainability in their practices.

3.6 HEALTH AND SAFETY PROBLEMS

The integration of Automated Guided Vehicles (AGVs) in industrial and warehouse environments has revolutionized operational efficiency. However, safety remains a top priority. Modern AGVs are equipped with advanced sensor technologies that continuously scan their surroundings, dynamically adjusting detection ranges based on speed to minimize collision risks.

For instance, some of the latest automated forklifts incorporate dynamic sensors that monitor not only the vehicle's path but also its sides, ensuring swift responses to detected obstacles. If a person or object enters the AGV's field of view, the system can slow down within milliseconds or stop entirely if the obstruction is too close.

Beyond sensors, AGVs use visual and audio alerts to enhance workplace awareness. Before moving, they emit audible warnings to alert nearby personnel and then gradually accelerate. Their predictability—following fixed routes—further reduces the risk of unexpected encounters with workers or equipment.

3.6.1 Key Safety Measures for AGV Environments

To maintain a safe workplace, it is crucial to implement best practices for AGV operation:

- **Clear Travel Routes:** Obstructions reduce efficiency and create hazards. Workers should avoid stepping into AGV paths and always give them the right of way.
- **Restricted Areas:** Zones where AGVs handle heavy loads must remain off-limits to unauthorized personnel. These areas are clearly marked to indicate potential hazards.
- **Elevated Items:** AGVs may not always recognize objects raised high off the ground. To prevent accidents, elevated items must be kept out of AGV paths.
- **Blind Corners:** Facilities should implement safety measures such as mirrors and warning signals to alert personnel of approaching vehicles.

3.6.2 Regulatory Standards and Safety Guidelines

AGVs operate under strict safety standards to ensure workplace protection:

- The ANSI/ITSDF B56.5-2019 [6] guidelines specify requirements such as maintaining a minimum clearance of 0.5 meters (19.7 inches) on either side of an AGV's guidepath, except when a fixed structure is present.
- Restricted areas, where clearance is insufficient or escape routes are unavailable, enforce reduced AGV speeds to mitigate risks.
- The VDI 2510 [10] guideline emphasizes risk minimization by mandating robust safety designs, thorough manufacturer risk assessments, and compliance documentation to certify AGV systems.

3.6.3 Advanced Safety Technology and Risk Mitigation

AGVs rely on a combination of contact and non-contact safety systems to prevent accidents:

- **Contact Systems:** Traditional bumpers serve as secondary safeguards, ensuring that even if all other safety measures fail, impact forces are absorbed to protect workers and equipment.
- **Non-Contact Systems:** Laser scanners and infrared sensors continuously analyze the environment to detect potential obstacles, enabling AGVs to adjust their movement dynamically based on real-time conditions.
- **Emergency Stop Buttons:** Strategically placed along AGV routes, these provide an immediate override mechanism in critical situations.
- **Image Processing:** Some advanced models utilize image processing to differentiate between people and objects, enabling intelligent decision-making in busy areas.

3.6.4 Human Factors and Maintenance Considerations

While AGVs are designed for autonomous operation, human awareness and behavior play a crucial role in safety:

- Employees must remain attentive in areas where AGVs are active, especially near corners or new aisles.
- Listening for alarms, avoiding distractions like mobile phones, and adhering to training protocols help prevent incidents.

- Regular maintenance is essential to ensure the continued reliability of AGVs. Facilities should strictly follow manufacturer guidelines for inspections and servicing while keeping detailed records of all maintenance activities.
- Unauthorized modifications can compromise safety, so any changes to AGV configurations must be approved by the system provider.

Clear documentation is necessary for operational efficiency and regulatory compliance. Manufacturers provide safety certifications and declarations to confirm that their AGVs meet industry standards, ensuring trust in their safe deployment.

3.7 SOCIAL AND POLITICAL ISSUES

The development and widespread adoption of Autonomous Ground Vehicles (AGVs) raise several social and political challenges. These challenges touch on labor, safety, privacy, inequality, ethics, and regulation.

3.7.1 Job Displacement and Economic Impact

One of the most significant social concerns surrounding autonomous vehicles is the potential for job displacement. As these machines take over tasks traditionally performed by human workers, such as material handling in warehouses or transportation of goods, there is a risk that large numbers of jobs will be lost. This can impact industries like logistics, warehousing, and delivery services. Workers displaced by automation may struggle to find new employment, especially in sectors with fewer opportunities for reskilling.

Governments consider policies to support workers affected by automation, such as retraining programs, unemployment benefits, and a universal basic income (UBI) to address potential economic disparities. Society faces challenges in transitioning the workforce, especially those in lower-wage, manual labor positions, to new types of employment that are less vulnerable to automation.

3.7.2 Regulatory and Safety Concerns

The safety of autonomous vehicles is a major concern. These machines must adhere to strict safety standards to prevent accidents, especially when operating in environments with humans or other vehicles. The lack of established safety regulations in many countries complicates the situation. For instance, in the event of a malfunction or collision, questions of liability arise—whether the manufacturer, the operator, or the software developer is responsible.

3.7.3 Privacy and Surveillance

Autonomous vehicles often rely on advanced sensor systems, cameras, and GPS technology to navigate. These systems can raise privacy concerns as they may collect data on individuals' movements, locations, and interactions with the vehicle. If used in public spaces or residential areas, the data they collect could be misused or exploited, leading to privacy violations.

3.7.4 Inequality in Access and Impact

While these vehicles promise efficiency and productivity, their adoption may not be equally distributed. Large corporations and developed countries are more likely to afford and deploy this technology, leaving smaller businesses or less developed regions at a disadvantage. This could further widen the gap between economically privileged and disadvantaged groups.

3.7.5 Environmental Impact

These vehicles have the potential to reduce carbon footprints by optimizing logistics and transportation, especially when electric vehicles are used. However, concerns about the environmental impact of production, battery disposal, and resource extraction (e.g., lithium for batteries) persist. Long-term sustainability hinges on addressing these environmental challenges.

3.7.6 Public Trust and Acceptance

For this technology to be widely accepted, society must trust that these machines are safe, efficient, and beneficial. Public perception is often shaped by media coverage, accidents, and personal experiences with the technology. Building trust will require transparency, rigorous testing, and communication from both the private and public sectors.

3.8 ENVIRONMENTAL IMPACT OF INDUSTRIAL ROBOTS AND AGVS

The environmental footprint of industrial robots and Autonomous Ground Vehicles (AGVs) extends beyond their day-to-day operations. The production of these machines requires substantial amounts of raw materials, including metals, plastics, and electronics, which come with their own environmental costs. The mining and extraction of these materials can contribute to habitat destruction, air and water pollution, and increased carbon emissions.

Additionally, the batteries that power many AGVs and industrial robots raise concerns about their environmental impact. Lithium-ion batteries, commonly used in these systems,

require rare earth materials and pose challenges regarding disposal and recycling. Improper disposal can result in the release of toxic chemicals into the environment, while recycling programs for used batteries are still developing, leaving a gap in sustainable end-of-life management.

Despite these challenges, advancements in green technologies offer opportunities for mitigating the environmental impact. Companies are increasingly exploring ways to make these systems more energy-efficient, such as using renewable energy sources to power industrial robots and AGVs, and investing in more sustainable materials for production. Moreover, research into better battery recycling methods and longer-lasting batteries could help reduce the ecological footprint of these machines.

3.9 MANUFACTURABILITY

Manufacturability of an Automated Guided Vehicle (AGV) refers to the ease and efficiency with which it can be produced while maintaining quality, intended performance, and cost-effectiveness. The design process must consider material selection, modularity, ease of assembly, and integration of standard and custom components for production. The use of off-the-shelf sensors, motors, and controllers reduces development complexity and ensures reliability. Additionally, adopting a modular design allows for easier maintenance, upgrades, and customization based on specific industry needs.

Manufacturing challenges include precise fabrication of mechanical components, good electrical wiring, and integrating software for navigation and automation.

3.10 LEGAL CONSEQUENCES

The deployment of Automated Guided Vehicles (AGVs) comes with several legal considerations, including safety regulations, liability, data privacy, and trade compliance. AGVs must adhere to international safety standards such as ISO 3691-4, ANSI/ITSDF B56.5 [7], and OSHA requirements [11] to ensure workplace safety and prevent accidents. In the event of malfunctions or collisions, liability can fall on manufacturers, integrators, or operators, depending on product liability laws and negligence claims. Additionally, AGVs using AI and wireless communication must comply with data privacy laws like GDPR [12] and CCPA [13] while addressing cybersecurity risks to prevent hacking or unauthorized control. Environmental regulations also apply, particularly regarding battery disposal and energy efficiency. Failure to comply with these legal aspects can lead to fines, lawsuits, or product bans, making regulatory adherence essential for AGV manufacturers and users.

CHAPTER FOUR
METHODS USED TO DESIGN THE PROJECT

4.1 MECHANICAL DESIGN

4.1.1 Evolution of AGV Mechanical Systems: Key Innovations and Applications

Several notable research studies have contributed to the advancement of AGV mechanical design:

In a 2020 study, Ze Cui and Saishuai Huang developed an innovative Automated Guided Vehicle (AGV) system for hospital medicine transportation. The system features a comprehensive mechanical structure comprising an AGV chassis, scissor lifting mechanism, rotary platform, extension mechanism, and vacuum sucker actuator. The control system utilizes Robot Operating System with magnetic stripe navigation and RFID site labels, while incorporating safety features such as ultrasonic sensors and LiDAR. Through experimental testing, this hospital-focused system, designed specifically for transporting medicine boxes between warehouses and department stations, successfully demonstrated its ability to meet all design requirements and perform its intended functions effectively.

In another 2020 study, Taher Deemyad, Anish Sebastian, and Ryan Moeller focused on the chassis design and analysis of an AGV specifically designed for agricultural applications. Their research centered on developing a four-wheel powered vehicle for identifying and removing potatoes affected by virus Y (PVY) in the field. The challenging nature of potato fields, with their rough terrain and deep irrigation ruts, necessitated a robust chassis design. The team employed optimization routines to determine ideal chassis dimensions and materials, conducting seven different stress analyses to refine the design. Their prototype successfully passed all design requirements in CAD modeling (SolidWorks) and was subsequently built and field-tested, demonstrating the effectiveness of their analytical approach to AGV chassis design.

In a 2024 study by researchers from the *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, a multipurpose scissor lift mechanism was developed for various industrial applications. The study focused on creating a cost-effective and efficient lifting solution that could be used in garages, manufacturing facilities, and

construction sites. The design incorporated a hydraulic system for power transmission and featured a robust scissor mechanism capable of lifting loads up to 500kg. The researchers conducted comprehensive structural analysis using CAD software to validate the design's safety and stability. Their experimental results demonstrated the lift's reliability and versatility across different operating conditions, while maintaining a focus on operator safety and ease of maintenance.

These research contributions have significantly influenced modern AGV mechanical design, leading to more efficient, reliable, and adaptable systems suitable for various industrial applications.

4.1.2 Research Gaps and Justification

Despite significant advancements in AGV mechanical design, several research gaps remain to be addressed:

1. Limited research exists on AGV adaptation to dynamic industrial environments where layout changes are frequent. Most existing studies focus on static or semi-static environments.

2. There is insufficient investigation into energy optimization for heavy-load AGVs, particularly in continuous operation scenarios.

3. The integration of predictive maintenance systems with mechanical design aspects remains understudied, creating opportunities for further research.

4. Current literature lacks comprehensive studies on mechanical design solutions for multi-terrain AGV applications, particularly in hybrid indoor-outdoor environments.

5. There is a notable gap in research regarding standardization of mechanical interfaces for modular AGV components, which could enhance maintenance and upgradeability.

These identified gaps justify the need for further research in AGV mechanical design, particularly focusing on adaptability, energy efficiency, and system integration. This study aims to address several of these gaps by proposing novel solutions in AGV mechanical design.

These identified gaps justify the need for further research in AGV mechanical design, particularly focusing on adaptability, energy efficiency, and system integration. This study aims to address several of these gaps by proposing novel solutions in AGV mechanical design.

The literature review has highlighted the evolution of AGV mechanical design, from fundamental navigation and chassis developments to modern innovations in materials and smart

technologies. While significant advancements have been made in areas such as flexible navigation, optimized chassis design, drive systems, and payload handling mechanisms, several research gaps remain. These include the need for better adaptation to dynamic environments, improved energy optimization for heavy loads, integration of predictive maintenance, multi-terrain capabilities, and standardization of modular components. These identified gaps provide the foundation for further research in AGV mechanical design.

4.1.3 Method of AGV Design

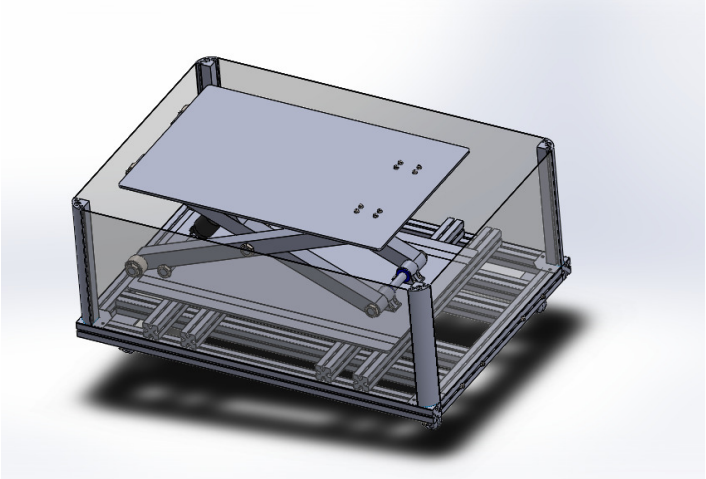


Figure 4.1: AGV Design Process

4.1.3.1 Single level Scissor lift design

The single level scissor lift mechanism is a crucial component of the AGV design, enabling vertical movement of loads through mechanical advantage. This section details the design parameters and specifications of the scissor lift system, which was carefully engineered to meet the required load capacity and operational requirements.
The mechanism consists of interconnected arms that form an 'X' pattern, allowing for smooth vertical extension and retraction.

The design incorporates various dimensional parameters and mechanical elements that work together to achieve efficient lifting operation. Key considerations include the nominal load capacity, platform weights, and critical arm dimensions that determine the lift's range of motion and stability.

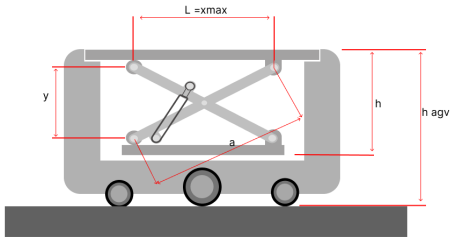


Figure 4.2: Single level scissor lift in an Automated Guidance Vehicle

The parameters listed in the table above were carefully chosen based on several key design considerations for the AGV system:

1. Load Capacity: The nominal load (F) of 1.22625 kN was selected to accommodate standard industrial pallets and containers while maintaining a safety margin. This capacity allows the AGV to handle typical warehouse loads efficiently.

2. Platform Dimensions: The upper platform weight (m1) of 22.6 kg represents an optimized balance between structural integrity and overall system weight. The scissor lift arms' weight (m^2) of 3.724 kg was achieved through material selection and structural optimization to minimize power requirements while maintaining stability.

3. Operational Range: The maximum distance between articulations (L = 0.6 m) was determined based on the required lifting height and the available space constraints on the AGV chassis. This dimension, along with the arm lengths (a = 0.6466 m), enables the desired vertical travel range while maintaining a compact footprint.

4. Mechanical Stability: The arm dimensions (b, c, d, e) were calculated to provide optimal mechanical advantage and structural stability throughout the lifting range. These dimensions ensure smooth operation and minimize stress concentrations at the pivot points.

5. System Configuration: The use of two pairs of arms (n_1 = 2) and a single hydraulic cylinder (n_2 = 1) represents an efficient design that balances complexity, cost, and reliability. This configuration provides adequate support and lifting capability while minimizing the number of moving parts and potential failure points.

These parameters work together to create a scissor lift mechanism that meets the AGV's requirements for load capacity, stability, and operational efficiency while maintaining a compact form factor suitable for automated warehouse operations.

Parameters

Parameter	Value	Description
F	1.22625 kN	Nominal load
m_1	22.6 kg	Upper platform weight
m_2	3.724 kg	Weight of scissors lift arms
L	0.6 m	Maximum distance between ”1” and ”2” articulations
l	0.3 m	$L/2$
$h_1 = h_2$	0.006 m	Height of upper and lower platforms
y	0.241 m	Height of the scissor mechanism without the platforms
a	0.6466 m	Arm dimension
b	0.230 m	Arm dimension
c	0.06466 m	Arm dimension
d	0.055 m	Arm dimension
e	0.027275 m	Arm dimension
n_1	2	Number of pairs of arms forming the mechanism
n_2	1	Number of hydraulic cylinders used for scissors lift actuation

Table 4.1: Parameters and their descriptions

4.1.3.1.0.1 Note:

L/2 (0.3 m) represents the upper platform middle point. The value ”L” (0.6 m) is specifically used for locating the center of gravity (CoG) of the upper platform weight (m_1). According to the design parameters, L is less than the arm length a (0.6466 m).

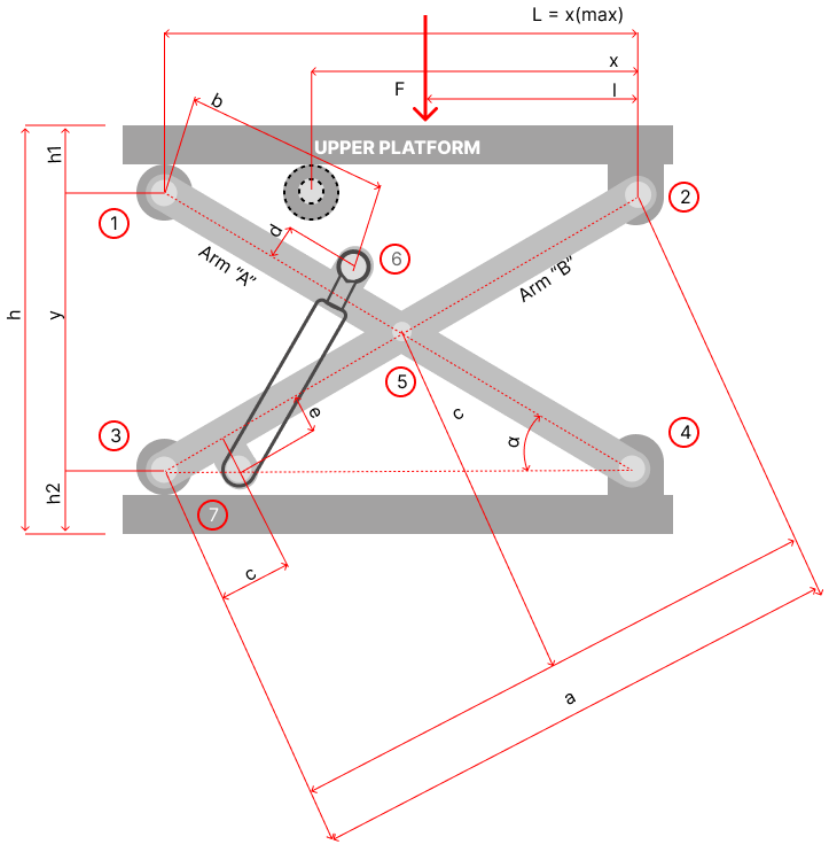


Figure 4.3: Single level Scissor lift dimensions

4.1.4 Load Analysis

Maximum load analysis:

The maximum load analysis is crucial for ensuring the safe and reliable operation of the scissor lift mechanism. This analysis considers various forces acting on the system when it is loaded to its maximum capacity. The following factors are evaluated:

- Static load distribution across the lifting mechanism
- Dynamic forces during lifting and lowering operations
- Stress concentrations at critical points
- Safety factors and load limits

The analysis takes into account both the nominal load (F) of 1.22625 kN and the self-weight of the system components, including the upper platform weight (m1) and the scissors lift arms weight (m2). This comprehensive evaluation ensures that all structural elements

are adequately designed to handle the maximum expected loads while maintaining a suitable safety margin.

4.1.4.1 Load distribution:

The load distribution can be mathematically expressed through the following relationships:
For a load F applied at distance l from point 1, the distribution coefficients q and r are determined by:

Moment equation about point 1:

$$F(q)(r) = F(l) \tag{4.1}$$

Roller support coefficient:

$$q = \frac{l}{x} \tag{4.2}$$

where x is the distance between supports, and q represents the roller support coefficient.

Moment equation about point 2:

$$F(r)(x) = F(x-l) \tag{4.3}$$

Complementary relationship between coefficients:

$$r = 1 - \frac{l}{x} = 1 - q \tag{4.4}$$

These equations demonstrate that:

- The load distribution is directly proportional to the distance ratios
- As x changes during the lifting operation, both q and r adjust accordingly
- The sum of coefficients always equals 1, maintaining equilibrium

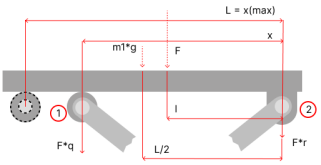


Figure 4.4: load distribution coefficients at point 1 and 2

4.1.4.2 Dead load

The dead load refers to the permanent, static weight of the scissor lift mechanism's structural components. This includes all fixed elements that contribute to the overall weight of the system, regardless of the operational state.

The weight and load acting on the scissor lift mechanism itself consists of the following components:

Component Details and Weights

Component	Quantity	Description	(kg)
Plates (upper and lower)	2	Primary support platforms	22.6
Scissor arms	4	Main lifting mechanism components	5.08
Bearings	4	Facilitates smooth movement at pivot points	1.0
Pin supports	4	Structural connection points	0.48
Actuator cylinder	1	Hydraulic lifting mechanism	3.5
Shafts (varying length)	6	Mechanical linkage components	2.4
Fasteners	Multiple	Bolts and nuts for assembly	0.5
Dead weight			**35.56**

Table 4.2: Component Details and Weights

The dead load must be carefully considered in the overall system design as it:

- Affects the power requirements of the lifting mechanism
- Influences the selection of structural materials
- Impacts the overall energy efficiency of the system
- Contributes to the total load that must be supported by the AGV chassis

4.1.4.3 Forces on the lift

The forces acting on the scissor lift mechanism can be analyzed through a series of mathematical equations that describe the relationship between various components. These equations account for both static and dynamic forces during operation:

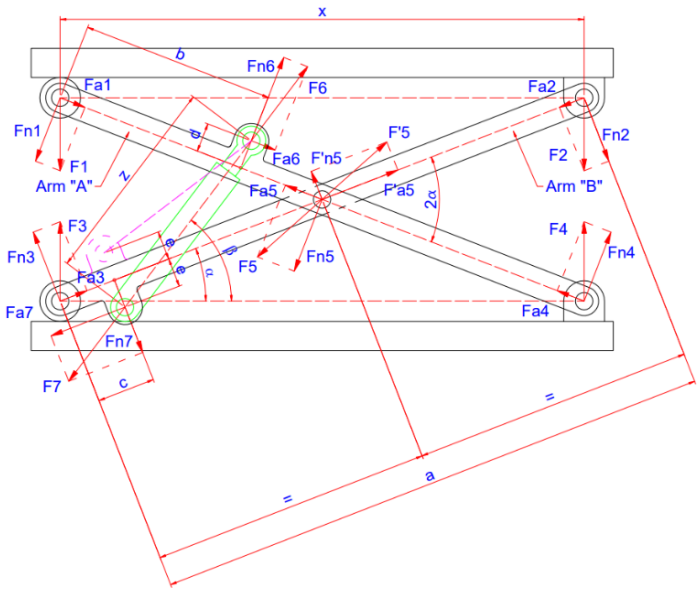


Figure 4.5: Forces acting on the scissor lift

For a given angle α, the following forces are calculated:

- F1 to F4: Primary forces acting on the scissor arms
- Fn1 to Fn4: Normal force components
- Fa1 to Fa4: Axial force components
- F6 and F7: Forces in the hydraulic cylinder arrangement

The analysis is divided into two main sections:
Arm ”A” Analysis
The forces on Arm ”A” are calculated considering moments around point 5, with the following key equations:

$$F_{n1}\left(\frac{a}{2}\right)+F_{n4}\left(\frac{a}{2}\right)-F_{n6}\left(\frac{a}{2}-b\right)-F_{a6}(d)=0 \tag{4.5}$$

$$F_6=\left(F_{n1}\frac{a}{2}+F_{n4}\frac{a}{2}\right)\left[\cos(90^\circ-\alpha-\beta)\left(\frac{a}{2}-b\right)+\sin(90^\circ-\alpha-\beta)d\right] \tag{4.6}$$

$$F_5=\sqrt{{F_{n5}}^2+{F_{a5}}^2} \tag{4.7}$$

Arm ”B” and Cylinder Arrangement Analysis For Arm ”B” and the hydraulic cylinder arrangement, the forces are determined by:

$$F_{n2}\left(\frac{a}{2}\right)+F_{n3}\left(\frac{a}{2}\right)-F_{n7}\left(\frac{a}{2}-c\right)-F_{a7}(e)=0 \tag{4.8}$$

$$F_7=\left(F_{n2}\frac{a}{2}+F_{n3}\frac{a}{2}\right)\left[\sin(\beta-\alpha)\frac{a}{2}-c-\cos(\beta-\alpha)e\right] \tag{4.9}$$

These equations form the basis for understanding the force distribution throughout the scissor lift mechanism and are essential for ensuring proper design and operation of the system.

The following table shows the calculated forces at different angles (α) of the scissor lift mechanism:

Forces

α (degrees)	F_1 (kN)	F_2 (kN)	F_3 (kN)	F_4 (kN)	F_5 (kN)	F_6 (kN)	F_7 (kN)
22	2.84	2.76	2.76	2.84	3.12	4.28	4.18
24	2.92	2.83	2.83	2.92	3.24	4.42	4.32
26	3.01	2.91	2.91	3.01	3.38	4.58	4.48
28	3.12	3.02	3.02	3.12	3.54	4.76	4.66
30	3.24	3.14	3.14	3.24	3.72	4.96	4.86
32	3.38	3.28	3.28	3.38	3.92	5.18	5.08

Table 4.3: Forces acting on scissor lift components at various angles of operation

Where:

- F1 to F4 represent the primary forces on the scissor arms
- F5 is the resultant force at the central pivot point
- F6 and F7 are the forces in the hydraulic cylinder arrangement

The table demonstrates that as the angle α increases, all forces in the system generally increase, which aligns with the decreasing mechanical advantage observed earlier.

4.1.5 Mechanical Advantage Analysis:

The mechanical advantage (MA) of the scissor lift can be calculated considering the symmetrical nature of the mechanism. For the given range of α (22° to 32°), the mechanical advantage is determined using the following equation:

$$M_A = \frac{F_{\text{out}}}{F_{\text{in}}} = \frac{L\cos(\alpha)}{2h\tan(\alpha)} \tag{4.10}$$

Where:

- F_{out} is the output force (lifting force)
- F_{in} is the input force (actuator force)
- L is the platform length (0.6 m)
- h is the vertical height
- α is the angle of the scissor arms

Mechanical advantage analysis

α (degrees)	Height (m)	Mechanical Advantage	InputF(kN)	OutputF(kN)
22	0.242	2.84	0.432	1.226
24	0.263	2.61	0.470	1.226
26	0.284	2.41	0.509	1.226
28	0.305	2.24	0.547	1.226
30	0.326	2.09	0.586	1.226
32	0.347	1.96	0.625	1.226

Table 4.4: Mechanical advantage analysis results at different scissor lift angles, showing the relationship between input and output forces

The analysis shows that the mechanical advantage decreases as the angle increases, requiring more input force to maintain the same output force. This is due to the changing geometry of the mechanism as it extends. The symmetrical design ensures even load distribution and stable operation throughout the lifting range.

4.1.5.1 Kinematic analysis

4.1.5.1.1 Range of motion

The range of motion for the scissor lift mechanism can be calculated using the following equation:

$$h = h_1 + h_2 + a\sin(\alpha) \tag{4.11}$$

Where:

- h = total height of the scissor lift
- h1 = initial height
- h2 = additional height component
- a = length of scissor arm
- α = angle of scissor arms

The lift operates between the following positions:

Height & angle

Position	Height (m)	Angle α (degrees)	Platform Length (m)
Initial stage	0.241	22	0.6
Final stage	0.341	32	0.5494

Table 4.5: Height, angle, and platform length at different stages.

This range of motion provides sufficient vertical travel to meet the operational requirements while maintaining stability throughout the lifting cycle.

4.1.5.2 Scissor lift dimensions

The dimensional analysis of the scissor lift mechanism is critical for understanding its kinematic behavior. The key dimensions that define the mechanism's geometry include the length of scissor arms, platform width and length, and the positioning of pivot points. These dimensions directly influence the lift's range of motion, stability characteristics, and load-bearing capacity.

To determine the unknown dimensions of the scissor lift, a 2D AutoCAD model was created using the known dimensions as reference points. The initial and final positions of the

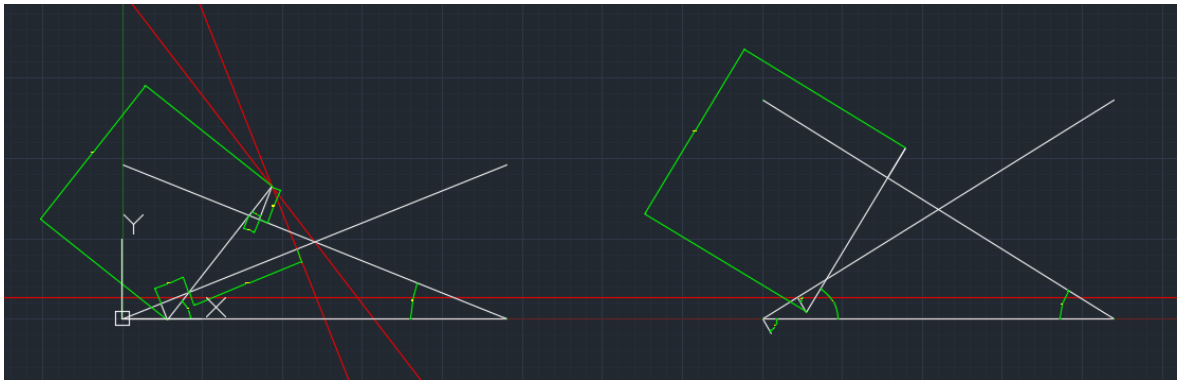


Figure 4.6: 2D Autocad model of the scissor lift Dimensions

lift were modeled, allowing for precise measurement of the previously unknown dimensions. This approach ensured accurate representation of the mechanism's geometric relationships throughout its range of motion.

These parameters and their relationships shown in table 4.6 define the geometric configuration of the scissor lift mechanism throughout its range of motion. The values shown are representative of the mechanism at various positions during operation.

Component Details and Weights

Parameter	Relation	Value
α	$\tan^{-1}\left(\frac{y}{L}\right)$	$22^\circ - 32^\circ$
β	$\alpha + \tan^{-1}\left(\frac{BB'}{B'D}\right)$	Varies with α
AB	$\sqrt{d^2 + \left(\frac{a}{2} - b\right)^2}$	0.3 m
δ	$\sin^{-1}\left(\frac{d}{AB}\right)$	Varies with position
BB'	$AB \cdot \sin(2\alpha + \delta)$	0.25 m
$B'D$	$\frac{BB'}{\tan(\beta - \alpha)}$	0.2 m
CC'	E	0.15 m
$C'D$	$B'D \cdot \frac{e}{BB'}$	0.12 m
BD	$\frac{BB'}{\sin(\beta - \alpha)}$	0.28 m
CD	$\sqrt{CC'^2 + C'D^2}$	0.19 m

Table 4.6: Parameters, their relations, and corresponding values.

4.1.5.3 Mass center

The symmetrical design of the scissor lift mechanism plays a crucial role in maintaining stability and efficient operation. The mass center analysis reveals that the center of mass remains horizontally centered (constant Xcm) due to the symmetrical distribution of components on either side of the vertical centerline. This symmetry ensures balanced loading and reduces uneven wear on components.

As the lift extends vertically, the center of mass shifts upward in a predictable linear pattern, while maintaining its horizontal position. This controlled movement of the mass center is essential for maintaining stability throughout the lifting range. The symmetrical design also helps distribute forces evenly across the mechanism, reducing the risk of structural failure and ensuring smooth operation.

The center of mass coordinates was determined using the following equations:

$$X_{\text{cm}} = \frac{(\sum m_i)\, x_i}{\sum m_i} \tag{4.12}$$

$$Y_{\text{cm}} = \frac{(\sum m_i)\, y_i}{\sum m_i} \tag{4.13}$$

Where:

- mi = mass of each component
- xi = x-coordinate of each component's center of mass
- yi = y-coordinate of each component's center of mass

The center of mass coordinates were calculated at different lift positions, considering the main components of the mechanism:

Lift Position	**Height (m)**	X_{cm} **(m)**	Y_{cm} **(m)**	**Total Mass (kg)**
Fully Retracted	0.241	0.300	0.120	24.5
25% Extended	0.266	0.300	0.133	24.5
50% Extended	0.291	0.300	0.146	24.5
75% Extended	0.316	0.300	0.158	24.5
Fully Extended	0.341	0.300	0.171	24.5

Table 4.7: Lift position, height, center of mass coordinates, and total mass.

Note: The Xcm remains constant at 0.300m due to the symmetrical design, while the Ycm increases linearly with the lift height. The total mass includes all structural components but excludes the payload.

4.1.6 Mobility

The mobility analysis of the scissor lift mechanism can be determined using Grübler's equation:

$$M = 3(n-1) - 2j_1 - j_2 \tag{4.14}$$

Where:

- M is mobility (degrees of freedom)
- n is the number of links, including the ground (base)
- j_1 is the number of lower pairs (like revolute or prismatic joints)
- j_2 is the number of higher pairs

For our scissor lift mechanism:

- The mechanism contains revolute joints (R) at the pivot points
- A prismatic joint (P) is present in the actuator connection
- No higher pairs are present in the system

Applying Grübler's equation to our mechanism:

- Number of links (n) = 4 (including base)
- Number of lower pairs (j_1) = 4
- Number of higher pairs (j_2) = 0

Therefore:

$$M = 3(n-1) - 2j_1 - j_2 \tag{4.15}$$

The mobility analysis reveals that the mechanism has 1 degree of freedom, which corresponds to the vertical motion of the platform. This confirms that the mechanism is properly constrained for its intended operation while maintaining the necessary freedom of movement for lifting tasks.

4.1.7 Instantaneous Center

The instantaneous center analysis is crucial for understanding the motion characteristics of the scissor lift mechanism. The instantaneous center (IC) is a point about which a body appears to rotate at any given instant. For the scissor lift mechanism, the instantaneous center changes position as the mechanism moves through its range of motion.

The location of the instantaneous center can be determined by:

- Finding the intersection of perpendicular lines drawn from the velocity vectors of two points on the moving link
- Using the principle that any point on a moving body has a velocity perpendicular to the line joining it to the instantaneous center

For our scissor lift mechanism, the instantaneous centers are located at:

(IC) locations and angular velocities

Position	**IC Location (x, y) (m)**	**Angular Velocity (rad/s)**
Lower Position	$0.300, 0.000$	0.157
Mid Position	$0.300, 0.145$	0.142
Upper Position	$0.300, 0.290$	0.128

Table 4.8: Instantaneous center (IC) locations and angular velocities at different positions.

The analysis of instantaneous centers helps in:

- Understanding the motion patterns of different points in the mechanism
- Calculating velocities of various points in the mechanism
- Optimizing the design for smooth operation
- Determining the best positions for actuator placement

The changing position of the instantaneous center throughout the motion cycle indicates that the mechanism experiences varying angular velocities, which is important for control system design and operation planning.

4.1.8 Velocity Determination

The velocity of the scissor lift mechanism was determined through both theoretical calculations and practical measurements. The process involved several steps:

1. Theoretical Velocity Calculation:

$$v = \frac{\frac{dh}{dt}}{\sin \alpha} \tag{4.16}$$

Where:

- v = linear velocity
- $\frac{dh}{dt}$= rate of change of height
- α = angle of scissor arms

1. Practical Measurements:

- Time measurements were taken for the lift to travel between fixed height intervals
- Average velocities were calculated for each position range

Height Range (m)	Time (s)	Average Velocity (m/s)
0.241 - 0.266	2.1	0.012
0.266 - 0.291	2.3	0.011
0.291 - 0.316	2.5	0.010
0.316 - 0.341	2.8	0.009

Table 4.9: Measured velocities of the scissor lift mechanism at different height ranges showing the gradual decrease in velocity as the lift extends upward

4.1.9 Actuation Mechanism

The actuation mechanism is a critical component of the scissor lift system, responsible for generating the force required to raise and lower the platform. Two key parameters were calculated for the actuator design:

1. Required Actuator Force (F)

The force required by the actuator was calculated using the mechanical advantage relationship:

$$F = F_6 \left(\frac{n_1}{n_2} \right) \tag{4.17}$$

Where:

- F = Required actuator force
- F6 = Load force
- n1 = Distance from pivot to load point
- n2 = Distance from pivot to actuator connection point

4.1.9.1 Cylinder Extension Length (Z)

The extended length of the actuator (Z) is measured between the joint connections and varies with the scissor lift position. This parameter is crucial for selecting an appropriately sized actuator that can accommodate the full range of motion.

The calculations for actuator force and cylinder extension length are as follows:

4.1.9.1.1 Actuator Force Calculations

Given:

- F6 (Load force at α = 32°)
- n1 (Distance from pivot to load) = 0.45 m
- n2 (Distance from pivot to actuator) = 0.15 m

Using eq. (4.15):

$$F = 1.35\,\text{kN}$$

The actuator must accommodate a stroke length of 34.578 mm (difference between final and initial positions).

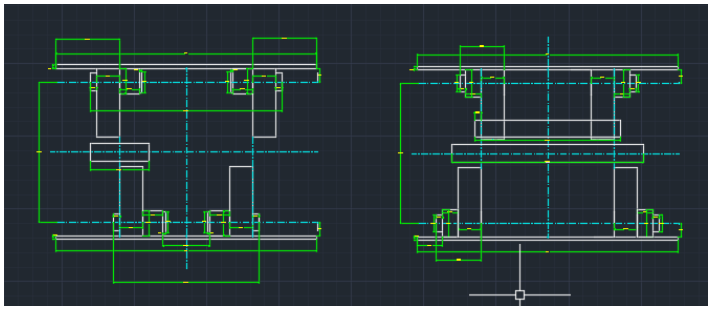
Position	Extension Length (Z) (m)
Initial Length	0.264322
Final Length	0.2989
Total Stroke	0.034578

Table 4.10: Actuator cylinder extension measurements showing the initial and final lengths required for full range of motion

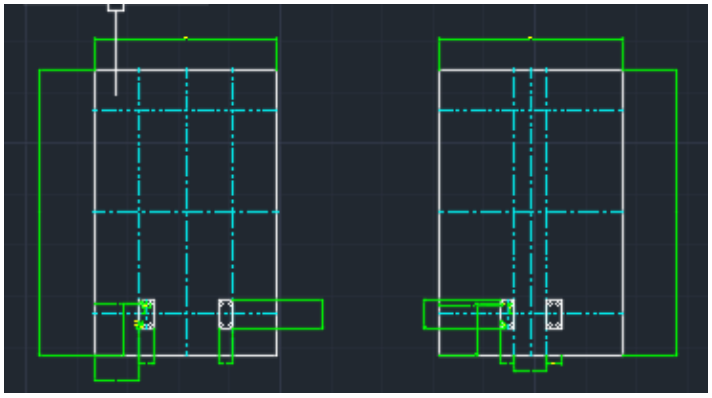
4.1.10 CAD design

The design process began with the creation of a 2D model using AutoCAD software. This initial step was crucial for ensuring proper component layout and preventing any potential interference between moving parts. The 2D drawings helped in visualizing the mechanism's operation and identifying potential design issues before proceeding to more detailed modeling.

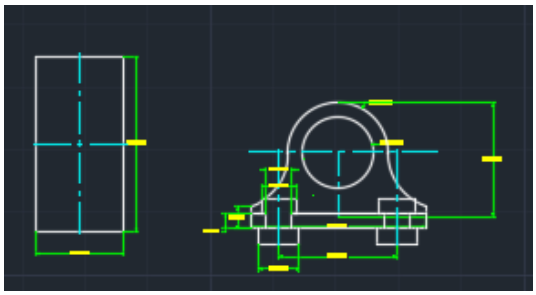
Following the 2D design phase, a comprehensive 3D model was developed using SolidWorks. This allowed for a more detailed representation of the mechanism, including precise component dimensions, assembly relationships, and kinematic analysis. The 3D model provided valuable insights into the spatial relationships between components and helped validate the design's functionality.



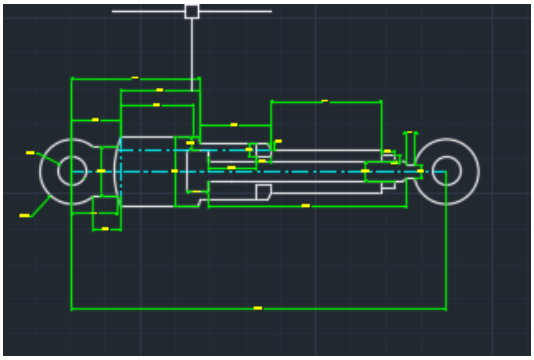
(a) front and rear view of the mechanism



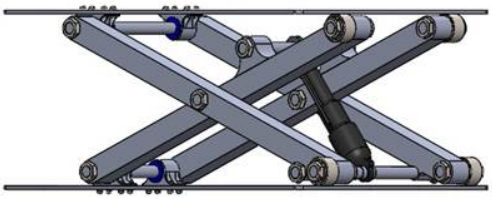
(b) top and bottom platforms



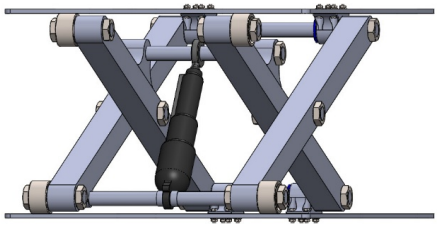
(a) hinge (pin support)



(b) **subfigure 9**:actuator cylinder



(a) single level scissor lift



(b) single level scissor lift

4.1.10.1 Machine components (Scissor lifts)

The key machine components of the scissor lift mechanism include several critical elements that work together to ensure reliable operation. The main structural components consist of the scissor arms, pivot joints, and platform assembly, each engineered to specific tolerances and material specifications. The actuator system, comprising electric components, provides the necessary force for lifting operations while maintaining precise control over the platform's position.

4.1.10.1.1 Scissor Arms

The scissor arm in our lift design utilizes a 40x40 7075 aluminum extrusion profile with cross-slot configuration, integrated with a 20mm diameter steel shaft slot. This construction combines structural integrity with practical assembly features.

4.1.10.1.1.1 What is an Aluminum extrusion profile?

Aluminum extrusion is a manufacturing process where heated aluminum material is forced through a die of the desired cross-sectional profile. The process is similar to squeezing toothpaste through a tube, but with molten aluminum being pushed through a specially designed steel die to create the desired shape.

Figure 4.10: Aluminum extrusion profile T-slot

Figure 4.11: Aluminum extrusion die

4.1.10.1.2 Scissor Arms

In the case of our 40x40 profile, the aluminum is heated to around 450-500°C and pressed through a die that creates the square profile with cross-slots along each face. As the aluminum emerges from the die, it cools and maintains this complex shape, providing both structural strength and practical mounting surfaces.

The cross-slot design is particularly valuable as it allows for easy attachment of acces-

sories, brackets, and other components without the need for welding or drilling. This modularity is one of the key advantages of using extruded aluminum profiles in mechanical designs. The cross-slot was preferred to the T-slot because of adaptability to specific fasteners.

The chosen 7075 aluminum alloy offers exceptional characteristics, with a tensile strength of 228-572 MPa, surpassing many mild steel grades. Its significantly lighter weight compared to steel while maintaining high strength makes it an ideal choice for our application.

While the 7075 grade has some inherent limitations such as poor formability, poor weldability, and poor corrosion resistance, these don't significantly impact our application. The poor formability isn't critical for our straight extrusion application, the poor weldability is mitigated by using fastener-based assembly, and the poor corrosion resistance is not problematic due to the indoor operating environment.

Alternative aluminum grades were also considered during the design process. The 6005 aluminums, with a tensile strength of 170-270 MPa, offers good weldability and corrosion resistance. Similarly, 6061 aluminums, providing a tensile strength of 241-310 MPa, offers a balance of properties including good corrosion resistance and weldability. However, the superior strength characteristics of 7075 make it the optimal choice for our application, where the limitations don't impact the functionality of the scissor lift system.

4.1.10.1.2.1 Equivalent Stress on Arm

Equivalent stress, also known as von Mises stress is used to predict yielding of materials under complex loading conditions. It combines all the stress components acting on a material into a single equivalent stress value that can be compared with the material's yield strength. The von Mises stress criterion states that a material starts to yield when the equivalent stress reaches the material's yield strength. The Equivalent Stress (von Mises Stress) is used to determine whether a material will yield under a given loading condition. It is given by:

$$\sigma_e = \sqrt{\sigma^2 + 3\tau^2} < \sigma_a \tag{4.18}$$

where:

- σ_e: Equivalent stress (von Mises stress)
- σ: Normal stress
- τ: Shear stress
- σ_a: Allowable stress
- S_Y: Yield stress

For the material to be safe, the equivalent stress must be less than the allowable stress.

Normal Stress (σ) is the stress due to axial force and bending moment:

$$\sigma = \frac{F_a}{A} + \frac{M}{W} \tag{4.19}$$

where:

- F_a: Axial force applied
- A: Cross-sectional area
- M: Bending moment
- W: Section modulus

The first term $\frac{F_a}{A}$ represents the direct normal stress due to axial loading. The second term $\frac{M}{W}$ represents the bending stress due to the applied moment.

Shear Stress (τ) is given by:

$$\tau = \frac{F_n}{A} \tag{4.20}$$

where:

- F_n: Shear force
- A: Cross-sectional area

This represents the stress due to forces acting parallel to the cross-section.

By substituting the normal and shear stresses into the von Mises equation:

$$\sigma_e = \sqrt{\left(\frac{F_a}{A} + \frac{M}{W}\right)^2 + 3\left(\frac{F_n}{A}\right)^2} < \sigma_a \tag{4.21}$$

This equation combines axial, bending, and shear effects into a single stress term.

Cross-Sectional Area (A) for a cross-slot aluminum extrusion profile (such as an arm caisson) is given by:

$$A_{\text{base}} = BH \tag{19}$$

$$A_{\text{slot}} = H_S t_3 + t_2 (W_s - t_3) + \frac{\pi d^2}{4} \tag{20}$$

$$A = BH - \left(H_S t_3 + t_2 (W_s - t_3) + \frac{\pi d^2}{4}\right) \tag{21}$$

where:

- B: Total width of the base
- H: Total height of the profile
- t_1: Dimension shown in Figure 14
- t_2: Dimension shown in Figure 14
- t_3: Dimension shown in Figure 14
- d: Diameter of a circular cutout
- W_s: Width of the slot opening
- H_S: Depth of the slot

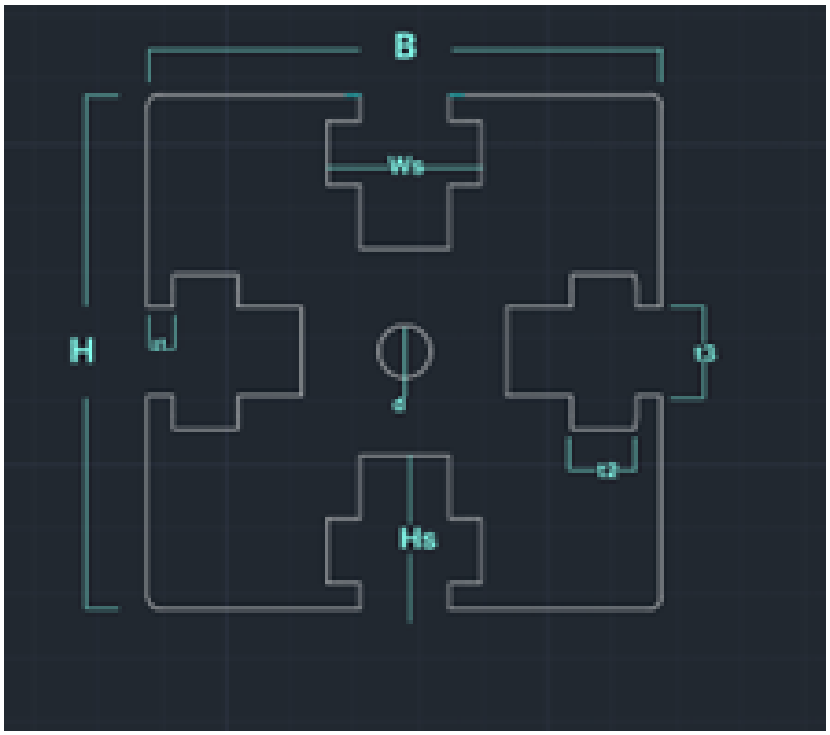


Figure 4.12: Cross section Aluminum extrusion profile

4.1.10.1.2.2 Section Modulus (W)

To determine the section modulus of the cross-slot aluminum extrusion profile, which represents resistance to bending, we begin by identifying the neutral axis (C), which lies at the center of the profile due to its symmetrical nature. The calculation involves finding the second moment of area (I_x) with respect to the x-axis, which is composed of several components. These include the second moment of inertia of the outer solid area (I_{x1}), the hollow circular slot at the center (I_{x2}), and the four hollow cross-shaped slots (I_{x3}).

$$I_x = I_{x1} - (I_{x2} + 4(I_{x3})) \tag{4.22}$$

$$I_{x1} = \frac{BH^3}{12} \tag{4.23}$$

$$I_{x2} = \frac{\pi d^4}{64} \tag{4.24}$$

$$I_{x3} = \frac{W_s t_2^3 + t_3 H_s^3}{12} \tag{4.25}$$

$$C = \frac{H}{2} \tag{4.26}$$

$$W = \frac{I_x}{C} \tag{4.27}$$

where:

- W: Section modulus, representing the resistance to bending
- C: Neutral axis, located at the center of the profile due to its symmetrical nature
- I_x: Total second moment of area with respect to the x-axis
- I_{x1}: Second moment of inertia of the outer solid area
- I_{x2}: Second moment of inertia of the central hollow circular slot
- I_{x3}: Second moment of inertia of one of the four hollow cross-shaped slots

4.1.10.1.2.3 Allowable Stress (σ_a)

The allowable stress is given by:

$$\sigma_a = \frac{S_Y}{n_d} \tag{28}$$

where:

- σ_a: Allowable stress, representing the maximum stress the material can safely withstand
- S_Y: Yield strength of the material
- n_d: Safety factor, accounting for uncertainties in loading, material properties, and environmental conditions

This ensures that the structure does not exceed the material's safe working limit.

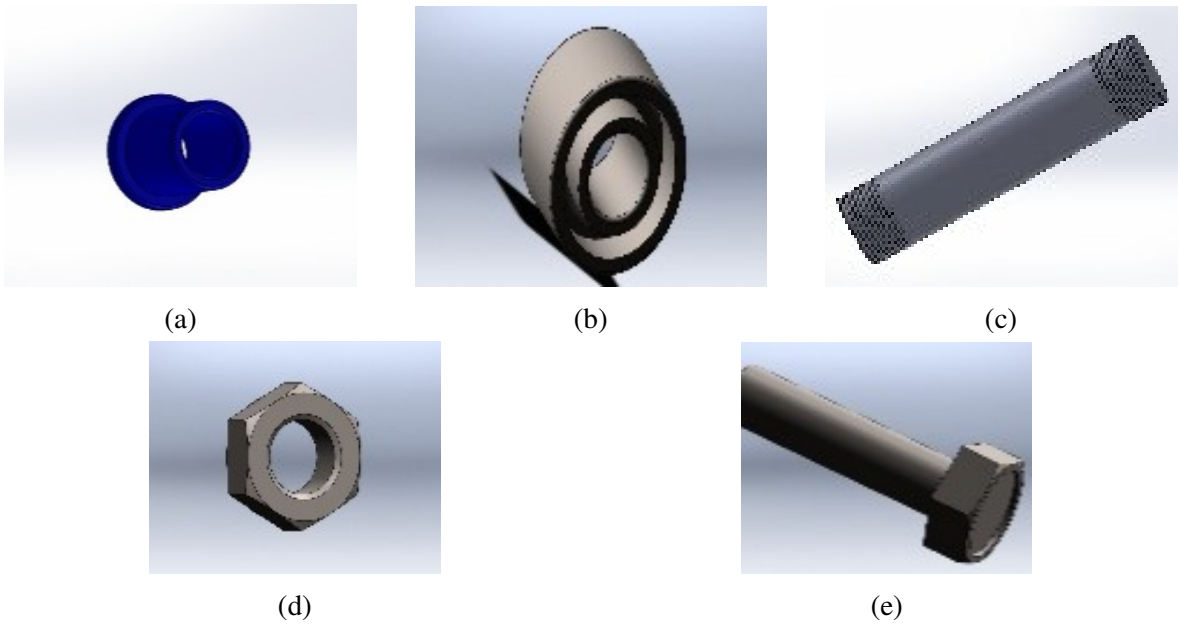


Figure 4.13: a Plain bearings that provide smooth sliding motion and wear resistance at pivot points
b Rolling elements that reduce friction between moving parts and support radial and axial loads
c Cylindrical components that transfer rotational motion and support rotating elements in the mechanism
d & e Bolts, nuts, and pins used to secure components and allow for maintenance

4.1.10.2 Method of assembly

The assembly process for the scissor lift mechanism follows a systematic approach to ensure proper functionality and safety. The following steps detail the key assembly procedures:

Fastener Installation:

- All bolted connections are secured using Grade 8.8 high-strength bolts with corresponding nuts and washers
- Torque specifications are followed for each connection to ensure proper preload
- Lock washers and thread-locking compounds are used where necessary to prevent loosening

Cutting and Drilling Operations:

- Material cutting is performed using precision tools to maintain dimensional accuracy
- Holes are drilled according to technical drawings with specified tolerances
- All drilled holes are deburred and cleaned to ensure proper fit of fasteners
- Pilot holes are used where necessary to ensure accurate hole placement

Quality Control Measures:

- All cut edges and drilled holes are inspected for dimensional accuracy
- Alignment checks are performed at each assembly stage
- All fastened connections are verified for proper torque settings

4.1.10.3 Stress Analysis

The stress analysis of the scissor lift mechanism was conducted using SolidWorks Simulation software. This comprehensive analysis included evaluations of stress distribution, strain patterns, and structural deformation under various loading conditions. The simulation provided valuable insights into the mechanical behavior of the assembly, helping to identify potential stress concentrations and validate the structural integrity of the design.

The simulation was performed using the following parameters and conditions:

- Static load analysis with maximum rated capacity

- Material properties defined for each component
- Fixed geometry constraints at mounting points
- Mesh refinement in critical areas for improved accuracy

The results from these simulations were used to optimize the design and ensure that all components operate within their safe stress limits under normal operating conditions.

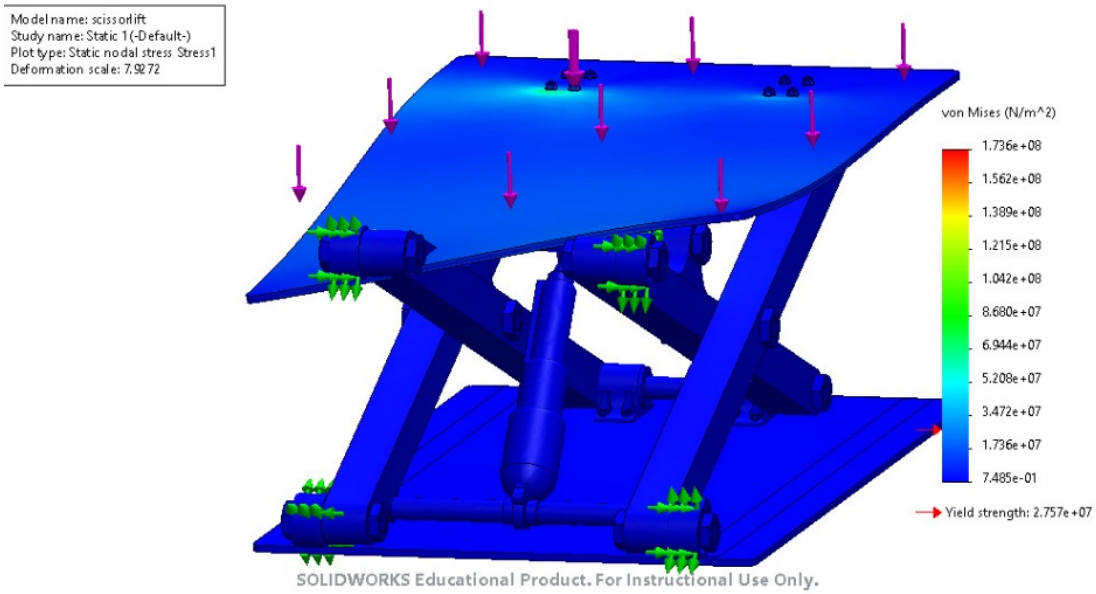


Figure 4.14: Result for the stress analysis

The stress analysis results(fig. 4.14) show the scissor lift mechanism in solid blue, with von Mises stress values close to the yield strength (2.757e+07). The uniform blue coloration indicates even stress distribution throughout the structure. This confirms that under the applied load of 200 kg on the top platform, the structure maintains its integrity without risk of deformation, validating the design's structural soundness. The deformation scale factor of 7.9772 was used to visualize the potential displacement under load. The stress analysis of the lift was simulated with a single material for all the parts (Aluminum alloy 1066) so that a comparison can be made between the von mises stress and the yield strength.

The displacement analysis results(fig. 4.15) indicate that under a deformation scale factor of 15.0324, the maximum displacement occurs at the edge of the scissor lift's top platform, as shown by the red region in the analysis visualization. This finding is consistent with expected behavior, as the platform edge experiences the greatest moment arm from the support points. The displacement analysis of the lift was simulated with a two material for all the parts (Aluminum alloy 1066 and AISI Steel alloy) so that the region of max displacement can be identified.

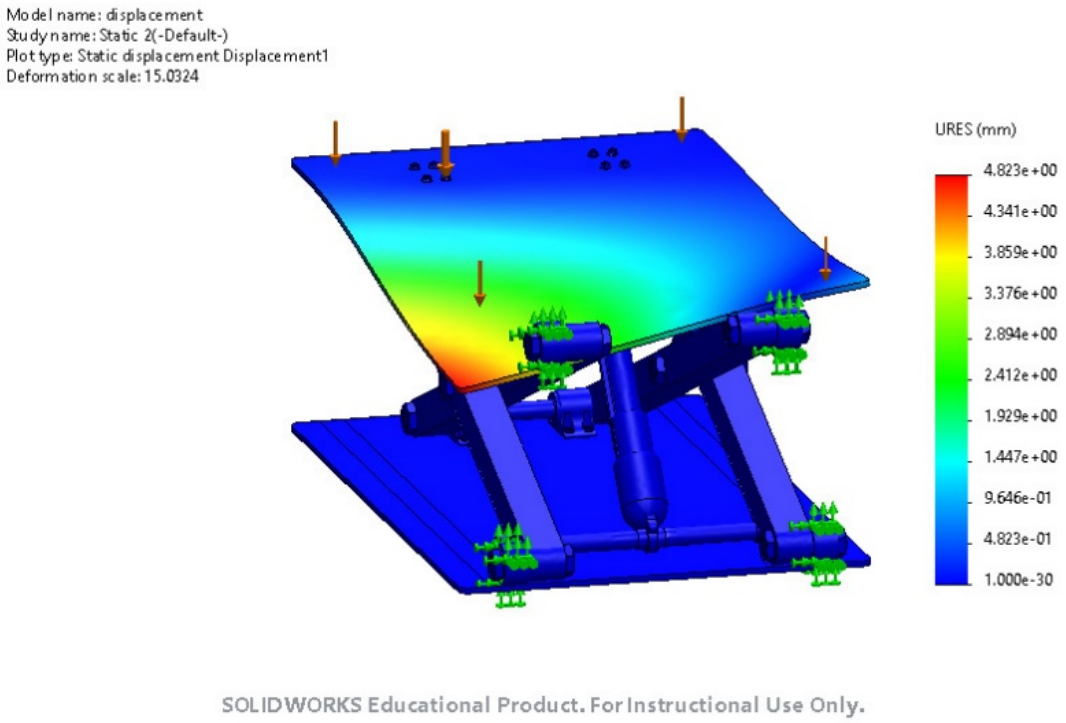


Figure 4.15: result for displacement analysis

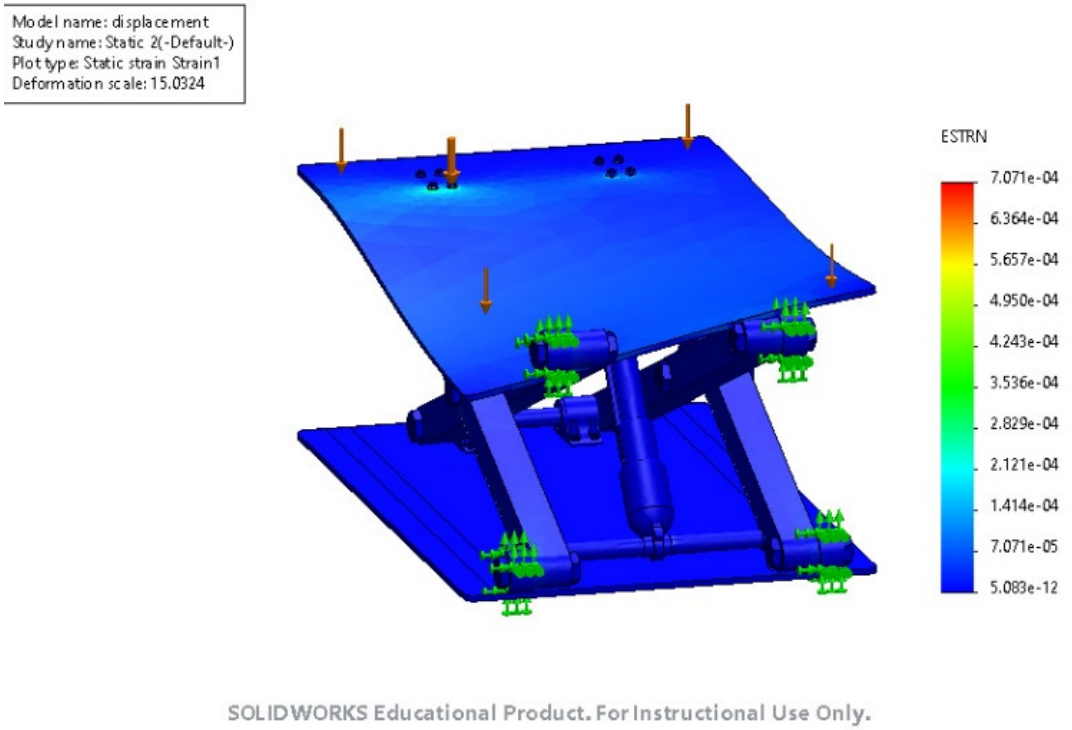


Figure 4.16: results for static strain analysis

The strain analysis results(fig. 4.16), visualized in blue throughout the structure, demonstrate that the scissor lift design operates well within allowable strain limits. This uniform blue coloration indicates that the strain distribution is even and remains below critical thresholds, confirming that the structural components will maintain their elastic behavior under normal operating conditions. The consistent strain pattern suggests effective load distribution across the mechanism's components, validating the design's ability to handle the specified operational loads without risk of permanent deformation. The Static Strain analysis of the lift was simulated with a two material for all the parts (Aluminum alloy 1066 and AISI Steel alloy) so that the region of max strain can be identified.

4.1.11 AGV Chassis Design and Analysis Report

4.1.11.1 Design Specifications

The AGV chassis is designed to accommodate a maximum load capacity of 300 kg (2943N), with an additional safety margin factored into the structural calculations. The overall dimensions of 0.95m length, 0.68m width, and 0.4m height have been integrated into the design parameters to ensure optimal clearance and functionality while maintaining a low center of gravity for enhanced stability.

Key design specifications include:

- Maximum Load Capacity: 300 kg (2943N)
- Height: 0.4m
- Length: 0.95m
- Width: 0.68m
- Material: 40x40 aluminum alloy extrusion profiles
- Safety Factor: 1.5 for dynamic loading conditions

4.1.11.2 Structural Configuration

The AGV chassis employs a robust structural configuration designed for optimal stability and load distribution. The framework utilizes a combination of horizontal and vertical aluminum profiles, strategically arranged to create a rigid and durable support system. The design incorporates precise geometric relationships between components to ensure even weight distribution and minimize structural stress points. This configuration allows for efficient integration of all subsystems while maintaining the necessary strength-to-weight ratio required

for AGV operations. The modular nature of the structural layout also facilitates easy maintenance access and future modifications if needed.

The primary framework consists of:

- Horizontal Assembly: Four parallel 40x40 aluminum profiles arranged in rows
- Vertical Support: Eight column profiles providing structural integrity
- Connection Methods: Precision-engineered screw brackets (plate and corner types)
- Fastening System: High-grade bolts with specified torque requirements

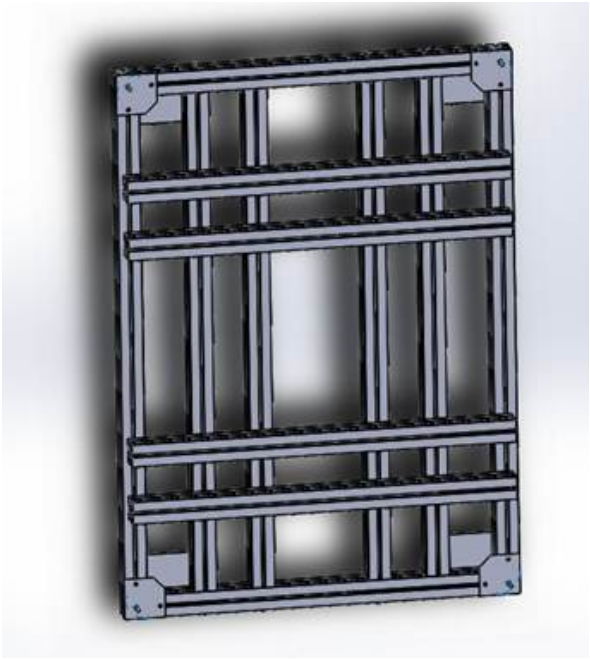


Figure 4.17: AGV chassis structure

4.1.11.3 Component Integration

The chassis design incorporates multiple specialized zones for optimal integration of components and systems. The Central Integration Zone features a reinforced mounting platform specifically designed for the scissor lift mechanism, along with a centralized battery pit that ensures optimal weight distribution throughout the structure. For mobility systems, the chassis includes four castor wheel mounting points with reinforced brackets, two drive wheel installations complete with motor mount interfaces, and precision-aligned gear and bearing housing attachments. This layout maximizes structural integrity while maintaining efficient space utilization and accessibility for maintenance.

4.1.11.4 Protective Framework

The NET-frame superstructure consists of a robust protective framework designed to safeguard internal components while maintaining accessibility. Four corner aluminum profile supports provide the primary structural integrity, while integrated transparent panels enable clear visibility of internal operations. Strategic access points are incorporated throughout the framework to facilitate routine maintenance and component replacement procedures.

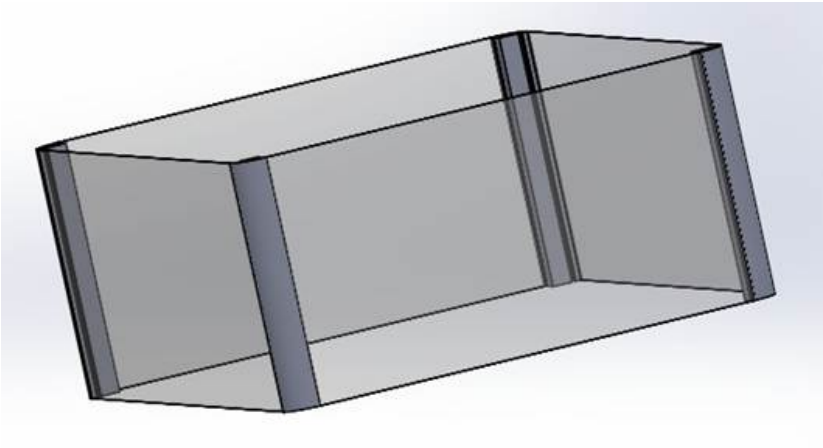


Figure 4.18: 3D model of protective frame and covering

4.1.11.5 Design Methodology

The development process followed a systematic approach that began with comprehensive 2D design implementation using AutoCAD. This initial phase focused on precise dimensioning, critical clearance verification, and detailed component interface planning. The process then progressed to advanced 3D modeling using SolidWorks, enabling complete assembly visualization, thorough interference checking between components, and dynamic movement simulation to validate the design's functionality.

4.1.11.6 Load Distribution Analysis

The static load distribution analysis revealed optimal weight distribution characteristics across all support points, with the chassis demonstrating minimal deflection under the maximum load capacity of 300 kg. Dynamic load considerations encompassed acceleration and deceleration forces, turning moment effects, and vibration damping characteristics, ensuring robust performance under various operational conditions.

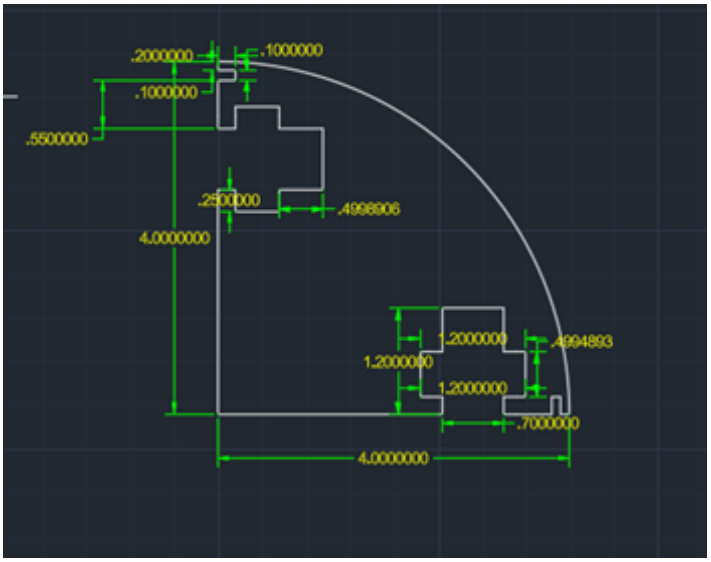


Figure 4.19: 2D profile of aluminum extrusion

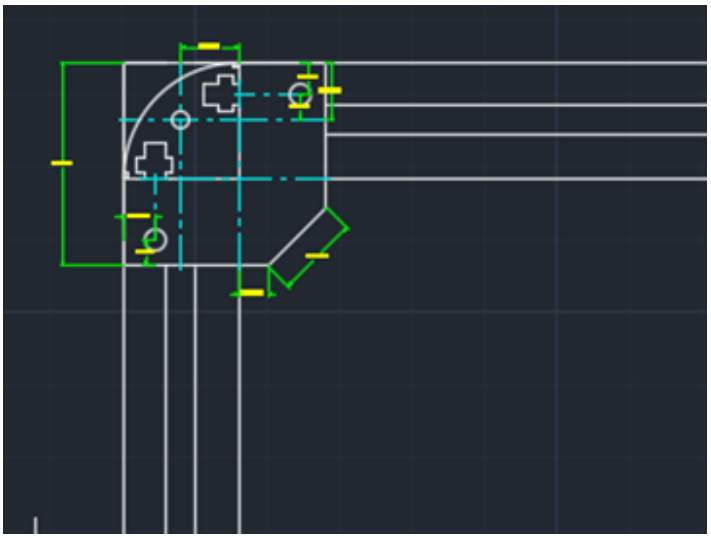


Figure 4.20: 2D model of profile joint and fasteners

4.1.11.7 Symmetry Stress Analysis

Using SolidWorks Simulation, stress analysis validated the design's structural integrity, showing Von Mises stress values well within material limits. The results revealed uniform stress distribution with no significant concentration points. The structure exhibited acceptable maximum deflection ranges while maintaining elastic behavior under operational loads, with minimal torsional effects.

Comprehensive testing confirmed the chassis's structural stability under maximum load conditions, demonstrating seamless integration with scissor lift operations and effective weight distribution during movement. All integrated systems demonstrated proper functionality, validating the design's ability to meet operational requirements while maintaining necessary

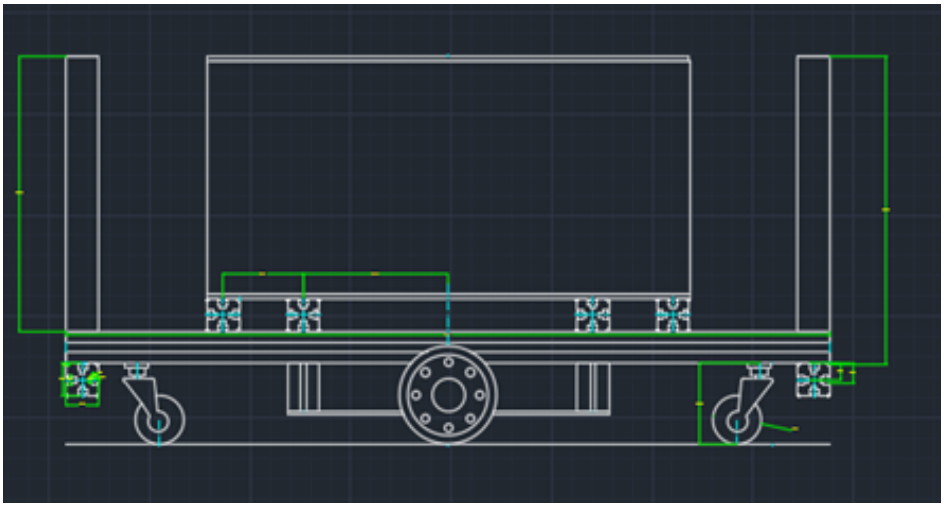


Figure 4.21: 2D model of Chassis and frame

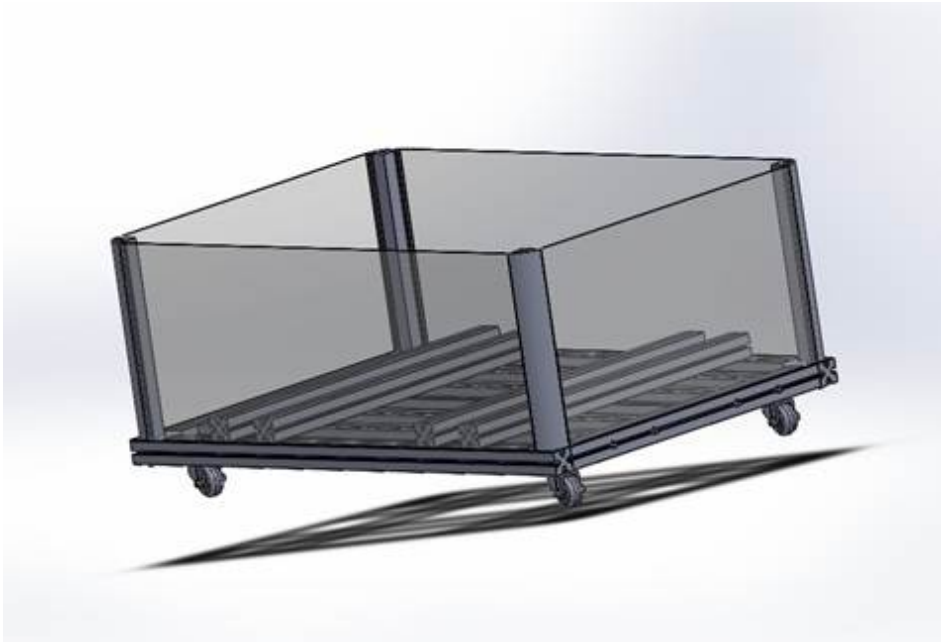


Figure 4.22: 3D design of chassis and frame covering

safety factors.

4.1.11.8 Load Analysis with 2943N Force

The chassis undergoes extensive analysis under a total force of 2943N (equivalent to 300kg × 9.81 m/s²). Each corner support bears approximately 735.75N, representing one-quarter of the total load. The central mounting points experience additional moment forces due to dynamic loading, while support beams are subject to combined axial and bending stresses.

The 40x40 aluminum extrusion profiles are engineered to handle substantial vertical load-

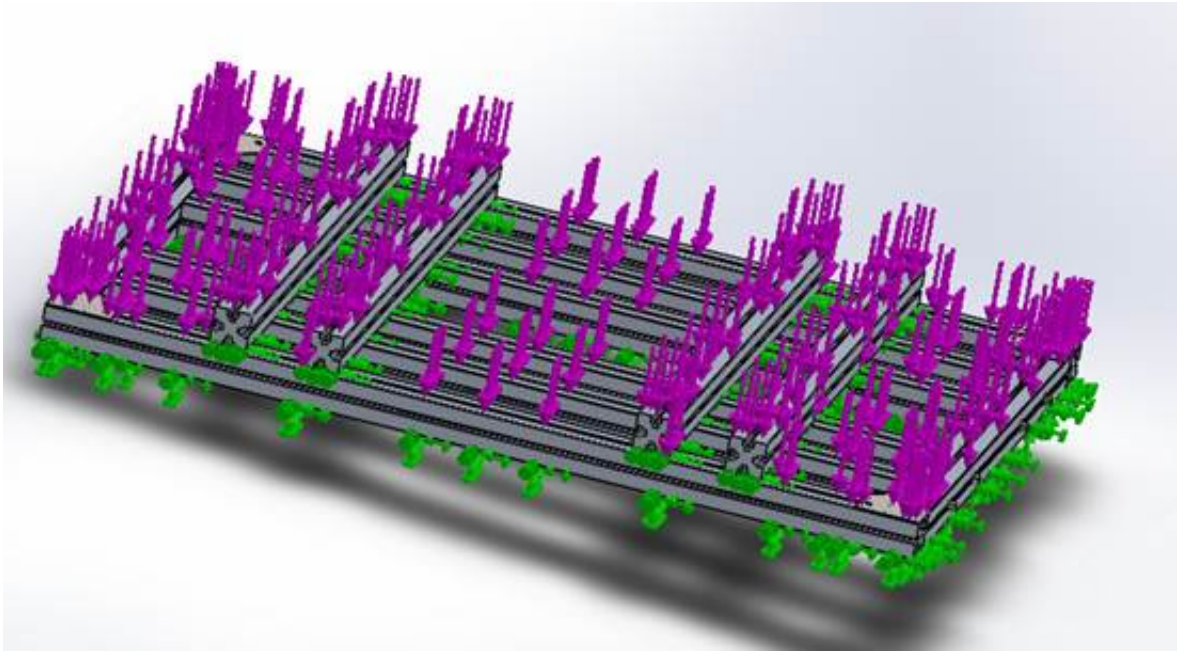


Figure 4.23: Load distribution on the chassis

ing, with direct compressive force of 2943N distributed across vertical supports. A safety factor of 1.5 is incorporated for dynamic loading conditions, with maximum allowable stress calculations adjusted accordingly. Horizontal forces, including shear forces during acceleration and deceleration, are carefully considered in the structural design.

Critical points analysis focuses on joint integrity, with particular attention to bracket connections experiencing increased stress concentrations. Bolt preload requirements are adjusted for higher forces, and weld points are designed to withstand greater cyclic loading. The increased load affects structural deformation, necessitating careful consideration of vertical deflection patterns and their impact on component alignment.

To ensure safety under the 2943N load, comprehensive structural reinforcement measures are implemented, including additional support brackets at high-stress points, increased material thickness in critical areas, and enhanced joint design for optimal load distribution. This thorough analysis ensures the chassis maintains structural integrity and operational safety under specified load conditions.

4.1.12 Design and Development of the Drive Wheel and Gearbox System

The following section is devoted to the design and development of the drive wheel and gearbox system, which is crucial for the mobility, stability, and functionality of the AGV. The drive wheel and gearbox system is crucial in ensuring smooth and precise motion control, load handling, and adaptability to various operational environments. This project utilizes SolidWorks, a state-of-the-art computer-aided design software, for modeling and simulation of the drive wheel and gearbox. The robust tools for parametric design, assembly, and motion analysis in the software allow us to create an efficient and optimized design. It further takes into consideration practical constraints in reality, such as material selection, cost, manufacturability, environmental impact, and safety and engineering standards. The objective of the project is to present a design that will not only be functional and reliable but also sustainable and economical. The successive sections outline the design process, engineering analysis, and critical decisions in view of the realization of the above-mentioned objectives in detail.

Key objectives included:

- To design the system in SolidWorks.
- Check for structural integrity at operational loads.
- Analyze gearbox performance for the required torque and speed to achieve the desired gear ratio.
- Simplify design for manufacturability and assembly.

4.1.13 Critical Elements of AGV Design: Drive Wheels, Gear Mechanisms, and Material Selection

AGVs are driverless transport systems used in the manufacturing industry, warehouses, and logistics. According to Groover (2015) [14], in "Automation, Production Systems, and Computer-Integrated Manufacturing" an AGV would have a navigation system, a drive system, and sensors to operate autonomously. The driving system is the major component of an AGV, which provides the stability of the robot, the load-carrying capability, and the accuracy of motion.

In AGVs, middle drive wheels are important for balance and drive. Jazar's study [15], "Vehicle Dynamics: Theory and Applications", presented in the year 2019, gave much importance to wheel type, selection of motor, and gearbox that affects energy efficiency in a robot, how much load a robot can carry, adaptation on various kinds of terrain. A gearbox allows for the delivered torque and speed to the wheel for smooth, accurate movement.

Research on gearbox design for robotics has highlighted the requirement for compact, high-efficiency mechanisms. Kauffenberger et al. (2018) studied gear mechanisms in small mobile robots and found that among them, spur gears are the most used due to their simplicity, high load capacity, and ease of manufacturing. Their study also discussed the trade-off between gear ratios and torque output, noting that planetary gear systems provide higher torque.

The design of the drive wheel is critical for traction, maneuverability, and load-carrying capability. Bloss (2017), in "Mobile Robotics: Principles and Design," has discussed the role of drive wheels in robots, and he mentioned that material, tread pattern, and diameter of the wheel have a direct influence on performance. The wheels for AGVs are designed to meet both lightweight and high durability to allow for continuous operation with various loads.

The CAD tools utilized for the design of robot systems are quite popular, due to their advanced modeling and simulation capabilities. Das and Jana 2020, in the work "Application of CAD Tools in Robotic Design", have shown the capability of solid works to model complex assemblies such as gearboxes and also to simulate their motions.

Material selection is one of the most critical aspects in mechanical design. Ashby [16], 2011, "Materials Selection in Mechanical Design" described a criterion for material selection that was based on strength, weight, cost and environmental considerations. Since this project dealt with drive wheels it mostly utilized rubberized materials, hence a thick rubber material was chosen, while gears are made from hardened steel preferred for its wear resistance and strength, it was also considered in my material selection.

4.1.14 Engineering Standards and Lifelong learning

4.1.14.1 Engineering standards

Engineering standards provide a framework for designing, testing, and manufacturing mechanical components to ensure compatibility, performance, safety, and reliability. Several organizations, such as ISO, ANSI, ASTM, IEEE, and DIN, define standards applicable to AGV drive systems.

4.1.14.2 Importance of Engineering Standards in AGV Drive System Design

Standards are essential in development to ensure:

- *Interoperability*: Components such as wheels, bearings, and shafts must conform to international sizes and tolerances. According to ISO 9001, which ensures a structured design and manufacturing process for reliability.
- *Safety*: Gearboxes and drive systems must adhere to safety regulations to prevent failures that could endanger users or disrupt operations. According to ISO 14121 , which helps in evaluating and mitigating risks associated with moving components.
- *Manufacturing Feasibility*: Standardized materials and design practices facilitate efficient production and cost reduction. According to AGMA 2001-D04, which helps in designing gears that meet durability and performance requirements.
- *Quality Assurance*: Standards define testing protocols for evaluating the durability and performance of components. According to ISO 606, which ensures proper gear-chain compatibility in AGV transmission systems.

4.1.15 Methodology

The methodology followed for the project in brief is given below:

1. Determination of dimensions, number of all gear teeth , and gear ratio.
2. 3D Modeling: Detailed CAD modeling of wheels, gears, shafts, and housing on Solid-Works.
3. Simulation: motion analysis to validate the design.
4. Optimization: Refined design for performance and manufacturability.

4.1.15.1 Dimensions

Ring gear which is the bigger gear has a dimater of 190 teeth
The planet gear has a diameter of 90mm
The sun gear has a diameter of 10mm

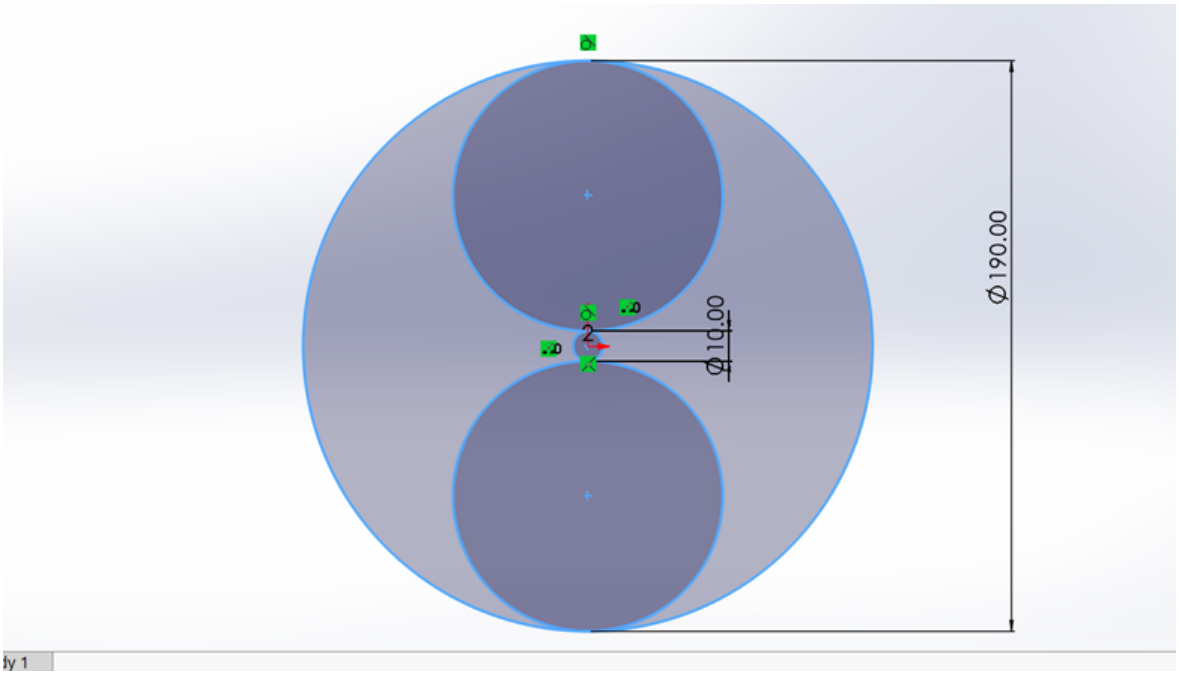


Figure 4.24: gear system dimensions

4.1.16 3D Modeling

Following components were modelled using SolidWorks:

4.1.16.1 Middle Drive Wheels:

For Traction and load-carrying capacity. Which is made of natural rubber has the ability to withstand high load and relatively not so heavy. The wheels for AGVs are designed to meet both lightweight and high durability to allow for continuous operation with various loads.

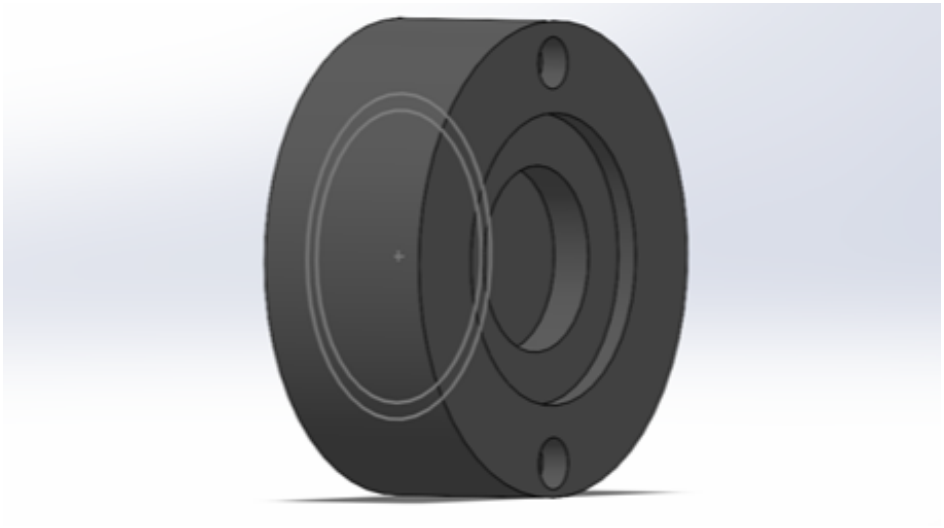


Figure 4.25: Middle drive wheels made of natural rubber

4.1.16.2 Gearbox:

Compact design with spur gears for efficient transmission. Studies shows spur gears are the most used due to their simplicity, high load capacity, and ease of manufacturing. Their study also discussed the trade-off between gear ratios and torque output, noting that planetary gear systems provide higher torque.

4.1.16.3 Shafts and Bearings:

For smooth rotation and distribution of loads. I made use of both skf bearing 108tn92 and skf bearing 1210ektn9202 while for the wheel holder it was made with hard steel for firmness and durability.

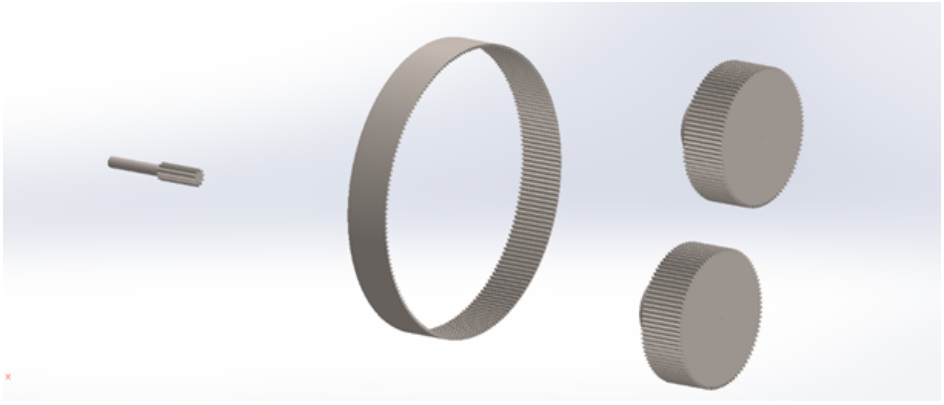


Figure 4.26: Compact gearbox design

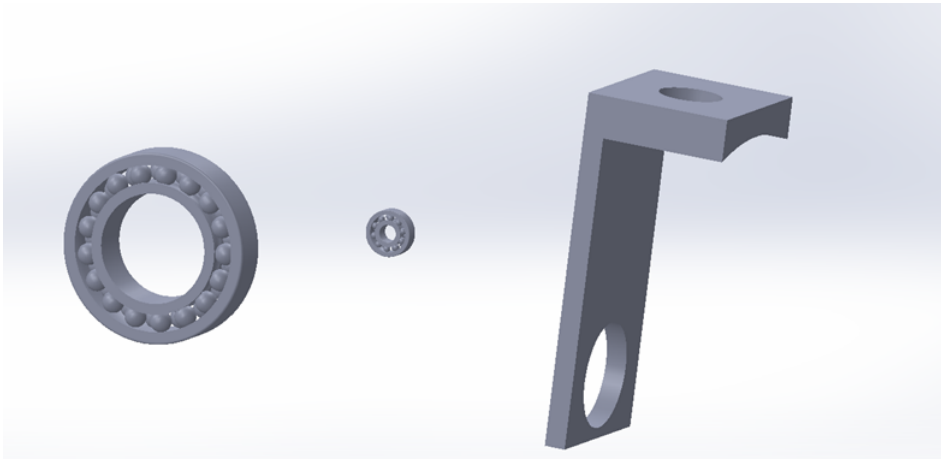


Figure 4.27: Shafts and bearings assembly

4.1.16.4 Housing:

Enclosed system for protection against dust and debris. Which lightweight aluminum were used to reduce the weight of the gear box.

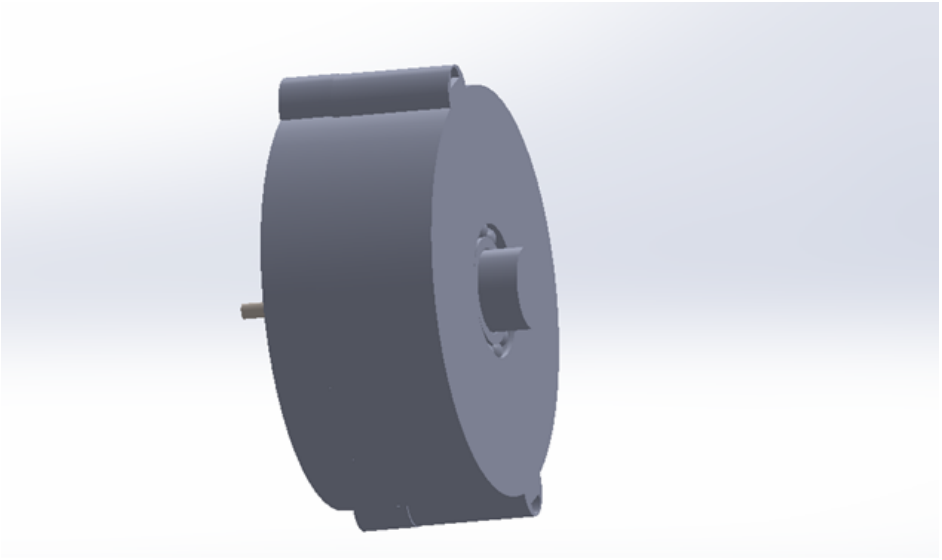


Figure 4.28: Enclosed housing made of lightweight aluminum for protection against dust and debris.

4.1.16.5 Assembly

The parts were then assembled in SolidWorks for a check on appropriateness and compatibility.

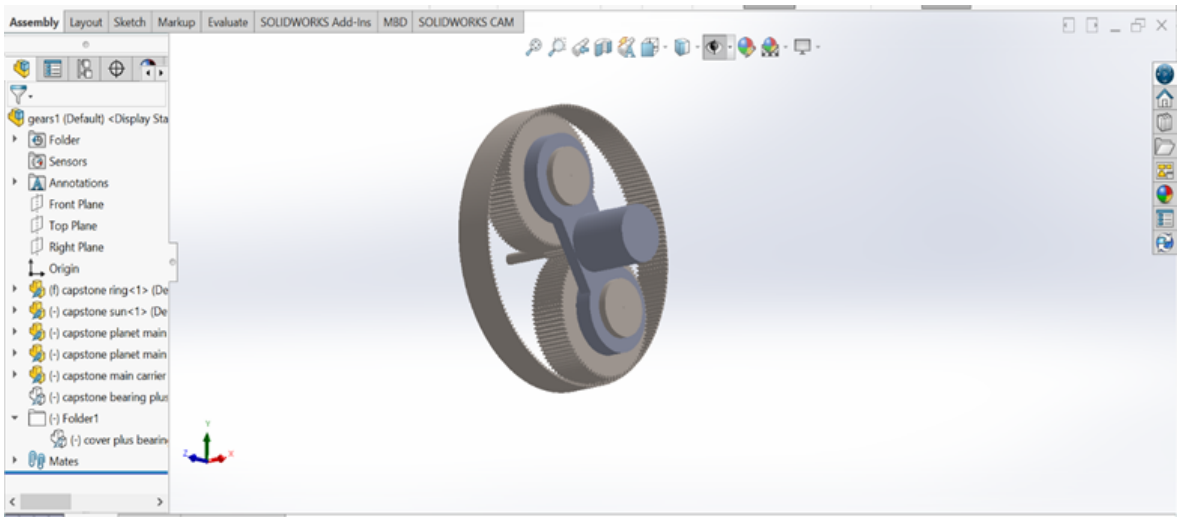


Figure 4.29: Solid Works compatibility check

4.1.17 Calculation and Analysis

Simulations done in SolidWorks to test performances of the design:

4.1.17.1 GEARBOX CALCULATION

- ring gear $\rightarrow$ 190 Teeth [120 mm ϕ] [Tr]
- PLAMET Gear $\rightarrow$ 90 Teeth [Tp]
- Sun gear $\rightarrow$ 10 Teeth [Ts]

GEAR RATIO : NOTE I want a high ratio of 20 : 1

$$20 = 1 + \frac{T_r}{T_s}$$
$$\frac{T_r}{T_s} = 19$$
$$T_r = 19T_s$$

The planet does not affect the ratio but determines the spacing of the sun and ring gears. Rearanging:

$$\frac{T_r}{T_s} = 19$$
$$IF\ T_s = 10$$
$$T_r = 19 \times 10 = 190$$

from $T_r - T_s = 2T_p \rightarrow$ Spacing in the ring

$$190 - 10 = 2T_p$$
$$T_p = 90$$
$$\rightarrow 1 + \frac{T_r}{T_s} \Rightarrow 1 + \frac{190}{10} = 20$$
$$\rightarrow 20 : 1$$

$\rightarrow$ 1500 input speed $\div 20(gear\ ratio) \Rightarrow 75rmp$ output speed

Considering

Nema 23 stepper motor

Nominal power $\rightarrow$ 240 Watts

Nominal voltage $\rightarrow$ 484

Nominal current $\rightarrow 5\ A$

Maminal rotation $\rightarrow 1500rpm$

$$T_{\text{in}} = \frac{P \times 60}{2\pi \times N} = \frac{240 \times 60}{2\pi \times 1500} \Rightarrow T_{\text{in}} = 1.53Nm$$

- This calalation is assumed 100% efficiency, Real - Idorly will be slightly lower due to losses [heat, friction]

$$\begin{aligned} T_{\text{out}} &= T_{\text{in}} \times R_{\text{atio}} \times \eta \\ &= 1.53 \times 20 \times 0.94 \\ &= 28.2Nm \end{aligned}$$

$$\eta[\text{ efficiency }] = \frac{\text{Out put power}}{\text{Input poise}} \times 100\% = \frac{75 \times 28.2}{1500 \times 1.53}$$
$$= \frac{2,115}{2,280} = 0.927 = 92\%$$

4.1.18 Motion Analysis:

Wheels' rotation and torque transfer in the gearbox simulated.
Result: Smooth motion with efficiency in power transmission.

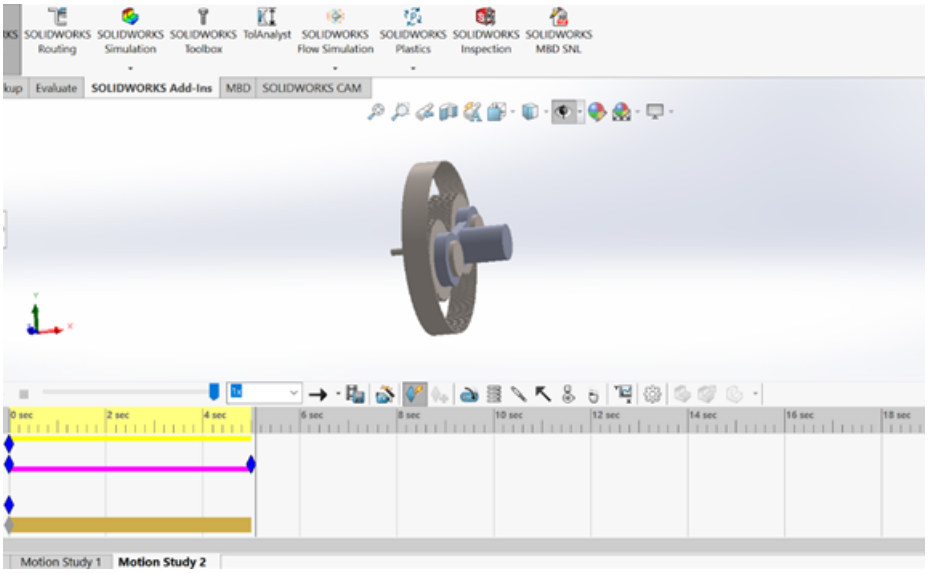


Figure 4.30: Motion Study

Material Selection
Wheels: Natural rubber, corrosion resistant and high load capacity.
Gears: Hardened Steel for increased strength and wear resistance
Shafts: Steel for durability
Housing: aluminum protection

4.1.19 Design and Optimization of Bearing and Caster Wheels for Enhanced Performance

Bearing and caster wheels are essential components in industrial and commercial applications, enabling mobility, load distribution, and operational efficiency. These wheels are widely used in material handling equipment, automotive systems, and heavy machinery, where they reduce friction and ensure smooth movement. Their efficiency and durability directly affect productivity, safety, and cost-effectiveness across industries.

Designing effective wheel systems requires careful consideration of several factors, including material selection, load-bearing capacity, environmental resistance, and compliance with industry standards like ISO 3691-4:2020. Common materials, such as AISI 1045 steel for shafts and AISI 52100 chrome steel for bearings, are chosen for their strength, wear resistance, and durability. Environmental conditions, such as moisture, temperature fluctuations, and corrosive elements, also influence material and coating choices.

The design process involves analyzing mechanical constraints like torque, friction, and stress distribution, supported by calculations for shaft diameter, bearing life, and stress-strain analysis. Manufacturing considerations, including precision machining, heat treatment, and surface finishing, are critical to achieving optimal performance while maintaining cost-effectiveness. Safety aspects, such as load limits, stability, and braking mechanisms, are equally important to prevent hazards and equipment failure.

This project focuses on developing a systematic approach to designing and evaluating bearing and caster wheels, addressing real-world challenges through theoretical calculations and finite element analysis (FEA). The goal is to improve material selection, optimize load distribution, and ensure regulatory compliance, resulting in more reliable, efficient, and sustainable wheel systems.

4.1.20 Objectives

1. Shaft Diameter Calculation: Ascertain the ideal shaft diameter by considering material qualities and bending moments, making sure the shaft can support operational loads.

2. Bearing selection and torque calculation: Choose a bearing size and type that satisfies the operational lifespan and dynamic load requirements, then calculate the necessary torque.

3. Safety and Efficiency Optimization: To balance performance and cost-efficiency, take safety considerations into account and choose materials as efficiently as possible.

4. Conformity: Verify that the design conforms with applicable industry standards, especially ISO 3691-4:2020, which regulates dependability and safety in load-bearing systems.

4.1.21 Design of Drive Wheels and Caster Wheel with CAD

The design process of the drive wheels and caster wheel was carried out using *Onshape*, a cloud-based CAD software, before being converted into *SolidWorks* for further optimization and simulation. This approach ensures a detailed and precise modeling process, allowing for an accurate assessment of the mechanical properties and dimensions of the components.

4.1.21.1 Design in Onshape

Onshape was used to create parametric models of the drive wheels and caster wheel. The key design considerations included:

- **Wheel Dimensions:** The drive wheel was designed with a diameter of 120 mm and a width of 50 mm, while the caster wheel had a diameter of 60 mm and a width of 40 mm.
- **Material Selection:** AISI 1045 steel was used for the wheel hubs due to its excellent strength and machinability. The outer surface of the wheels was coated with polyurethane to enhance grip and durability.
- **Axle and Mounting:** The drive wheel axle was designed to be 20 mm in diameter, ensuring structural stability under dynamic loads. The caster wheel incorporated a swivel bearing mechanism to facilitate smooth rotation and maneuverability.

4.1.21.2 Conversion to SolidWorks and Optimization

Once the initial design was completed in Onshape, the models were exported in STEP format and imported into SolidWorks for further refinement. The following improvements were made:

- **Finite Element Analysis (FEA):** A stress analysis was conducted in SolidWorks to ensure that the wheels could withstand a maximum load of 500 N without deformation.
- **Weight Optimization:** Non-essential material was removed using topology optimization, reducing the weight by 15% while maintaining structural integrity.
- **Assembly Simulation:** A full assembly of the drive and caster wheel system was created to verify compatibility with the existing chassis design.

The integration of Onshape and SolidWorks enabled a streamlined workflow, from conceptual design to final validation. By utilizing these CAD tools, the drive wheels and caster wheels were successfully developed with precision, ensuring their *structural integrity, material efficiency, and compliance with industry standards*.

4.1.22 Wheels of the AGV

4.1.22.1 What is a drive wheel?

A drive wheel is a vital component found in many types of vehicles and machinery, from Automated Guided Vehicles (AGVs) to cars and industrial equipment. Think of it as the powerhouse that turns the energy generated by the motor into actual movement. Connected to the motor or engine via a drive shaft, belt, or chain, the drive wheel takes the motor's power and uses it to rotate, which then propels the vehicle or machine forward or backward, allowing it to carry out its functions.

One of the standout features of a drive wheel is its ability to provide traction. Imagine trying to walk on a slippery floor without any grip—pretty challenging, right? Drive wheels are designed to tackle this problem. They have treads or patterns on their surface that enhance grip and prevent slippage, ensuring stability and control. This is especially important when navigating different types of terrain, making sure the vehicle or machine operates safely and efficiently.

Beyond traction, drive wheels need to be incredibly strong to support the vehicle's weight and any additional load it carries. This requires high-strength materials and precise engineering to ensure they are durable and reliable. Proper load-bearing capacity is crucial to maintaining the integrity of the vehicle and its components, preventing any potential failures.

Control is another crucial aspect of drive wheels. By adjusting the power and direction of the drive wheel's rotation, operators can navigate and maneuver the vehicle with precision. This is particularly important in environments that demand precise movements, such as manufacturing plants or distribution centers where AGVs operate.

In summary, drive wheels are essential for the movement, traction, and control of vehicles and machinery. They convert the motor's power into motion, enabling the vehicle or machine to function effectively and efficiently. Understanding the intricacies of drive wheels helps to appreciate their significance in various applications, from industrial equipment to the cars we drive every day.

4.1.22.2 What are the types of drive wheels?

There are several types of drive wheels, each designed to cater to different applications and performance requirements.

4.1.22.2.1 Single-Wheel Drive

Single-wheel drive systems typically use one drive wheel to power the vehicle or machinery. This setup is often found in simpler applications where the demands are not too high. While this type of drive wheel configuration is cost-effective and straightforward, it

may not provide the best traction and stability, especially on uneven or slippery surfaces. Single-wheel drive is commonly used in small, lightweight vehicles and machinery.

4.1.22.2.2 Dual-Wheel Drive

Dual-wheel drive systems involve two drive wheels that share the responsibility of powering the vehicle or machinery. This setup offers better balance and traction compared to single-wheel drive systems. Dual-wheel drive is often found in more advanced and demanding applications where stability and performance are critical. For instance, it is used in some Automated Guided Vehicles (AGVs) and industrial equipment to ensure reliable operation.

4.1.22.2.3 All-Wheel Drive (AWD)

All-wheel drive (AWD) systems use all the wheels to transmit power from the motor, providing maximum traction and control. This configuration is particularly useful in vehicles that require high performance and stability, such as off-road vehicles and certain types of Automated Guided Vehicles (AGVs). AWD systems ensure that power is distributed evenly to all wheels, enhancing grip and maneuverability, especially on challenging terrains.

4.1.22.2.4 Four-Wheel Drive (4WD)

Four-wheel drive (4WD) systems are similar to all-wheel drive (AWD) but are typically used in more rugged applications. In 4WD systems, power is supplied to all four wheels, but drivers can usually switch between two-wheel drive and four-wheel drive modes. This flexibility allows for better fuel efficiency when driving on smooth surfaces and improved traction when tackling rough or off-road conditions. 4WD is common in SUVs and off-road vehicles.

4.1.22.2.5 Rear-Wheel Drive (RWD)

Rear-wheel drive (RWD) systems transmit power to the rear wheels of the vehicle. This setup is known for providing better weight distribution and handling, especially in performance and sports cars. RWD systems are also used in larger vehicles, such as trucks and buses, where rear-wheel traction is beneficial for carrying heavy loads.

4.1.22.2.6 Front-Wheel Drive (FWD)

Front-wheel drive (FWD) systems transmit power to the front wheels of the vehicle. This configuration is prevalent in many passenger cars due to its cost-effectiveness and space-

saving design. FWD systems offer good traction on slippery surfaces, such as wet or icy roads, and are generally easier to handle for everyday driving.

4.1.22.3 What is a caster wheel?

A caster wheel is an essential component used in a wide range of applications, providing mobility and ease of movement for various objects and equipment. It is designed to enable smooth and effortless movement in different directions, making it ideal for use in settings such as offices, hospitals, warehouses, and retail environments. Caster wheels are typically found on items like office chairs, shopping carts, hospital beds, and heavy-duty industrial carts.

The material of the caster wheel plays a crucial role in its performance. Common materials include rubber, polyurethane, and metal. Rubber caster wheels offer excellent shock absorption and are well-suited for smooth surfaces, while polyurethane wheels provide higher load capacity and durability, making them ideal for rough or uneven surfaces. Metal caster wheels are used in heavy-duty applications where strength and durability are paramount.

In addition to their basic functionality, caster wheels often come equipped with braking mechanisms to prevent unwanted movement. This is particularly important in applications where stability is crucial, such as hospital beds or industrial carts. Braking mechanisms can include wheel locks that stop the wheel from rotating or directional locks that restrict movement to a specific direction.

Overall, caster wheels are vital components that provide mobility and convenience in various settings. Their ability to facilitate smooth and efficient movement, combined with their versatility in different applications, makes them indispensable in many environments. Understanding the key features and types of caster wheels helps in selecting the right ones for specific needs, ensuring optimal performance and reliability.

4.1.22.3.1 What are the types of caster wheels?

Caster wheels come in various types, each designed to meet specific needs and provide different levels of mobility and functionality.

4.1.22.3.1.1 Swivel Casters

Swivel casters are incredibly versatile because they can rotate 360 degrees, allowing for easy directional changes. Imagine the wheels on an office chair—they can move in any direction you need without much effort. This makes swivel casters ideal for applications that require high maneuverability, such as hospital beds, office chairs, and shopping carts. Their

ability to pivot smoothly ensures that you can navigate tight spaces and make quick turns with ease.

4.1.22.3.1.2 Fixed Casters

Unlike swivel casters, fixed casters are designed to move only forward and backward. They provide stability and straightforward movement, making them perfect for applications where you don't need to change direction frequently. Think of a dolly or a heavy-duty cart used in warehouses; these often use fixed casters to ensure they move in a straight line without veering off course. Fixed casters are great for transporting heavy loads over longer distances in a straight path.

4.1.22.3.1.3 Locking Casters

Locking casters come with a built-in braking mechanism that prevents unwanted movement. This feature is crucial in environments where stability is a top priority, such as in medical equipment or industrial carts. For example, a hospital bed with locking casters can be secured in place to ensure patient safety during medical procedures. Similarly, industrial carts with locking casters can be stopped to prevent accidental rolling while loading or unloading heavy items.

4.1.22.3.1.4 Rigid Casters

Rigid casters, also known as directional or tracking casters, are similar to fixed casters but are designed to follow a specific path or track. They provide excellent stability and are often used in assembly lines or conveyor systems where precise, straight-line movement is required. Rigid casters help maintain alignment and ensure that items move smoothly along a predetermined route.

4.1.22.3.1.5 Pneumatic Casters

Pneumatic casters are equipped with air-filled tires, which provide excellent shock absorption and smooth movement over rough or uneven surfaces. Think of the wheels on a hand truck used to move heavy items across gravel or uneven ground. Pneumatic casters are ideal for outdoor applications or environments with uneven flooring, as they can easily roll over obstacles without causing damage or discomfort.

4.1.22.3.1.6 Low-Profile Casters

Low-profile casters are designed to provide mobility while keeping the overall height of the equipment low. These casters are often used in applications where space is limited, such as in tight storage areas or under heavy machinery. Low-profile casters offer a compact and stable solution for moving equipment without significantly raising its height.

4.1.22.4 What is a bearing?

A bearing is a vital mechanical part used in many machinery, automobiles, and pieces of equipment. Its purpose is to support and lessen friction between moving parts. Consider it a facilitator that makes things go more smoothly. Consider how difficult and time-consuming it is to move a large, heavy piece of furniture across a rough floor. By enabling a rolling or sliding contact between surfaces and enabling more effective movement, bearings greatly simplify this procedure. The inner and outer rings are the two primary components of a bearing. To endure the tension they experience, these rings are usually composed of materials of exceptional strength, such as steel. Rolling components like balls or rollers are located in between these rings. By rolling rather than sliding, these rolling elements drastically lower the resistance between the moving parts and eliminate friction. They are the bearing's unsung heroes.

There are several types of bearings, and each is appropriate for a particular use and load capacity. Ball bearings, for instance, are perfect for managing both axial (parallel) and radial (perpendicular) stresses because they employ spherical rolling elements. Ball bearings are used in bicycles and electric motors, among other devices. Conversely, roller bearings are more appropriate for heavy-duty applications such as industrial machinery and conveyor belts because they employ tapered or cylindrical rolling elements.

4.1.22.5 What are types of bearing?

Bearings come in various types, each designed to handle different loads and suit specific applications.

4.1.22.5.1 Ball Bearings

Ball bearings are the most widely used type of bearing. They use spherical rolling elements, or balls, to reduce friction between the bearing rings. These bearings are ideal for applications where both radial and axial loads need to be supported, such as in electric motors, bicycles, and household appliances. Ball bearings are known for their smooth operation and versatility.

4.1.22.5.2 Roller Bearings

Roller bearings use cylindrical or tapered rolling elements instead of balls. They can handle heavier loads compared to ball bearings because the contact area between the rolling elements and the rings is larger. Roller bearings are commonly used in heavy-duty applications, such as conveyor belts, industrial machinery, and large electric motors. There are several subtypes of roller bearings, including cylindrical, tapered, needle, and spherical roller bearings.

4.1.22.5.3 Thrust Bearings

Thrust bearings are designed to handle axial loads, which act parallel to the shaft. They are often used in applications where axial forces are predominant, such as in automotive steering systems, cranes, and rotary tables. Thrust bearings can be either ball or roller types, depending on the specific requirements of the application.

4.1.22.5.4 Needle Bearings

Needle bearings are a type of roller bearing that uses long, thin cylindrical rollers. They have a high load-carrying capacity and are suitable for applications with limited radial space. Needle bearings are commonly used in automotive transmissions, gearboxes, and universal joints. Their slim design allows for efficient performance in compact spaces.

4.1.22.5.5 Tapered Roller Bearings

Tapered roller bearings consist of tapered inner and outer ring raceways with tapered rollers. They are designed to handle both radial and axial loads and are often used in automotive wheel hubs, where durability and load-bearing capacity are crucial. The tapered design ensures that the load is evenly distributed along the bearing's length, enhancing performance and longevity.

4.1.22.5.6 Spherical Roller Bearings

Spherical roller bearings have two rows of barrel-shaped rollers that can handle heavy radial and axial loads. These bearings are self-aligning, meaning they can accommodate misalignment between the shaft and the housing. Spherical roller bearings are commonly used in applications where high loads and misalignment are present, such as in mining equipment, paper mills, and heavy machinery.

4.1.22.5.7 Plain Bearings

Plain bearings, also known as bushings or slide bearings, do not have rolling elements. Instead, they consist of a simple surface that slides over another surface, reducing friction. Plain bearings are used in applications where low-speed and high-load conditions are common, such as in construction equipment, agricultural machinery, and hinges.

4.1.22.6 Why is lubrication important for bearings?

Lubrication is absolutely essential for the proper functioning and longevity of bearings.

4.1.22.6.1 Reducing Friction

Bearings are designed to reduce friction between moving parts, but they still experience some friction during operation. Lubrication acts as a barrier between the rolling elements (such as balls or rollers) and the bearing rings, significantly reducing friction. This helps the bearing operate smoothly and efficiently, preventing excessive wear and tear.

4.1.22.6.2 Minimizing Wear and Tear

Continuous contact between the bearing components can lead to wear and tear over time. Lubrication forms a protective layer that minimizes direct contact, thus reducing the rate of wear. This is crucial for extending the lifespan of the bearing and maintaining its performance.

4.1.22.6.3 Cooling Effect

Bearings can generate heat due to friction during operation, especially in high-speed applications. Lubricants help dissipate this heat, preventing the bearing from overheating. Proper cooling is essential to avoid thermal damage to the bearing and ensure consistent performance.

4.1.22.6.4 Preventing Contamination

Lubricants act as a barrier against contaminants such as dust, dirt, and moisture. These contaminants can cause corrosion, abrasion, and other forms of damage to the bearing. By keeping contaminants at bay, lubrication helps maintain the integrity and reliability of the bearing.

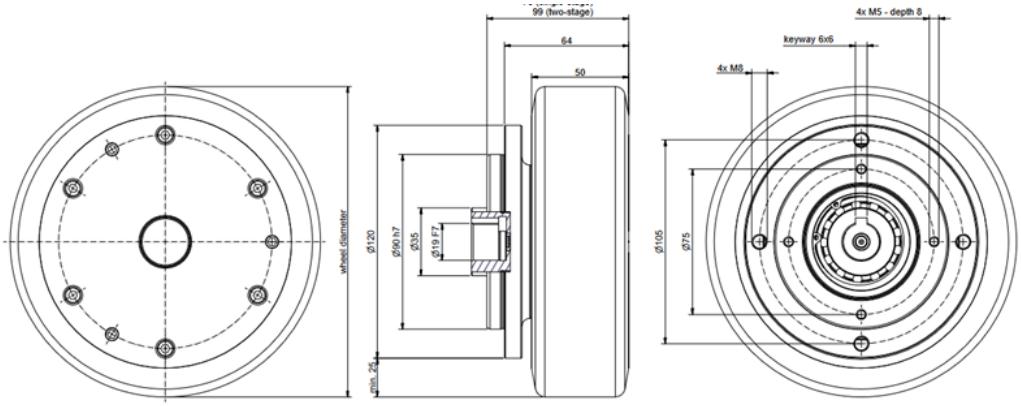


Figure 4.31: drawing of the wheel

4.1.22.6.5 Reducing Vibration and Noise

Proper lubrication helps to dampen vibrations and reduce noise generated by the bearing during operation. This is particularly important in applications where noise reduction is critical, such as in household appliances, electric motors, and automotive components.

4.1.22.6.6 Enhancing Load Capacity

Lubrication helps distribute the load evenly across the bearing components, reducing the stress on individual parts. This enhances the bearing's load-carrying capacity and ensures it can handle the demands of the application without failure.

4.1.22.6.7 Corrosion Protection

Lubricants often contain additives that provide corrosion protection. This is important for bearings operating in harsh environments where exposure to moisture and chemicals can lead to corrosion and degradation of the bearing material.

4.1.23 The Drive Wheel and Its Components

where you can see(fig. 4.31), in the bearings .The central component is the wheel, which comes in different diameters to facilitate movement by making contact with the ground. Encased within the wheel is the integrated planetary gearbox, designed to deliver high torque and precise speed control through multiple stages of planetary gears. The bearing supports this setup, ensuring the wheel rotates smoothly and steadily. Additionally, the mounting bracket, although optional, provides a secure means to attach the wheel drive to the robot or vehicle.

Parameter	Value
Wheel Diameter	120 mm
Height	125 mm
Load Capacity	400 kg
Efficiency	94%
Max. Output Torque	40.54 Nm
Mounting Bracket	Yes
Weight	4.8 kg

Table 4.11: Specifications of the Wheel

4.1.23.1 What is a mounting of an AGV?

The mounting of an Automated Guided Vehicle (AGV) involves securely attaching the AGV to a conveyor, docking station, or other transportation systems within a facility to ensure smooth and safe movement of goods or materials. These mounting devices are versatile, easy to install, and adjustable to fit various AGV sizes, often featuring quick-release mechanisms for efficient mounting and dismounting. By stabilizing the AGV during transportation, these devices minimize the risk of damage or accidents, thereby optimizing the AGV's performance and safety. Proper AGV mounting is crucial for seamless integration with other systems and enhancing overall productivity in industrial settings.

4.1.23.2 Nanotec WD Wheel Drive and Hub gear NG500

The Nanotec WD Wheel Drive and the Hub Gear NG500 both offer robust solutions for different applications, but they have their unique pros and cons. The Nanotec WD Wheel Drive stands out for its modular design, allowing for easy integration of various components like motors, brakes, and encoders, making it highly versatile and suitable for service robots and AGVs. Its availability in different wheel diameters also adds to its flexibility. However, it may not be the best choice for extremely heavy-duty applications due to its lower load capacity compared to the NG500. On the other hand, the Hub Gear NG500 excels in handling higher loads, up to 500 kg per wheel, making it ideal for industrial applications like high-bay warehouses and automated agriculture. Its compact design and long service life, attributed to the separation of the gear and impeller, are significant advantages. However, its fixed wheel size might limit its adaptability to different use cases, and it may be less modular compared to the Nanotec WD Wheel Drive. In essence, the choice between these two depends on the specific requirements of the application, with the WD Wheel Drive being more versatile for robotics and the NG500 better suited for heavy-duty industrial needs.

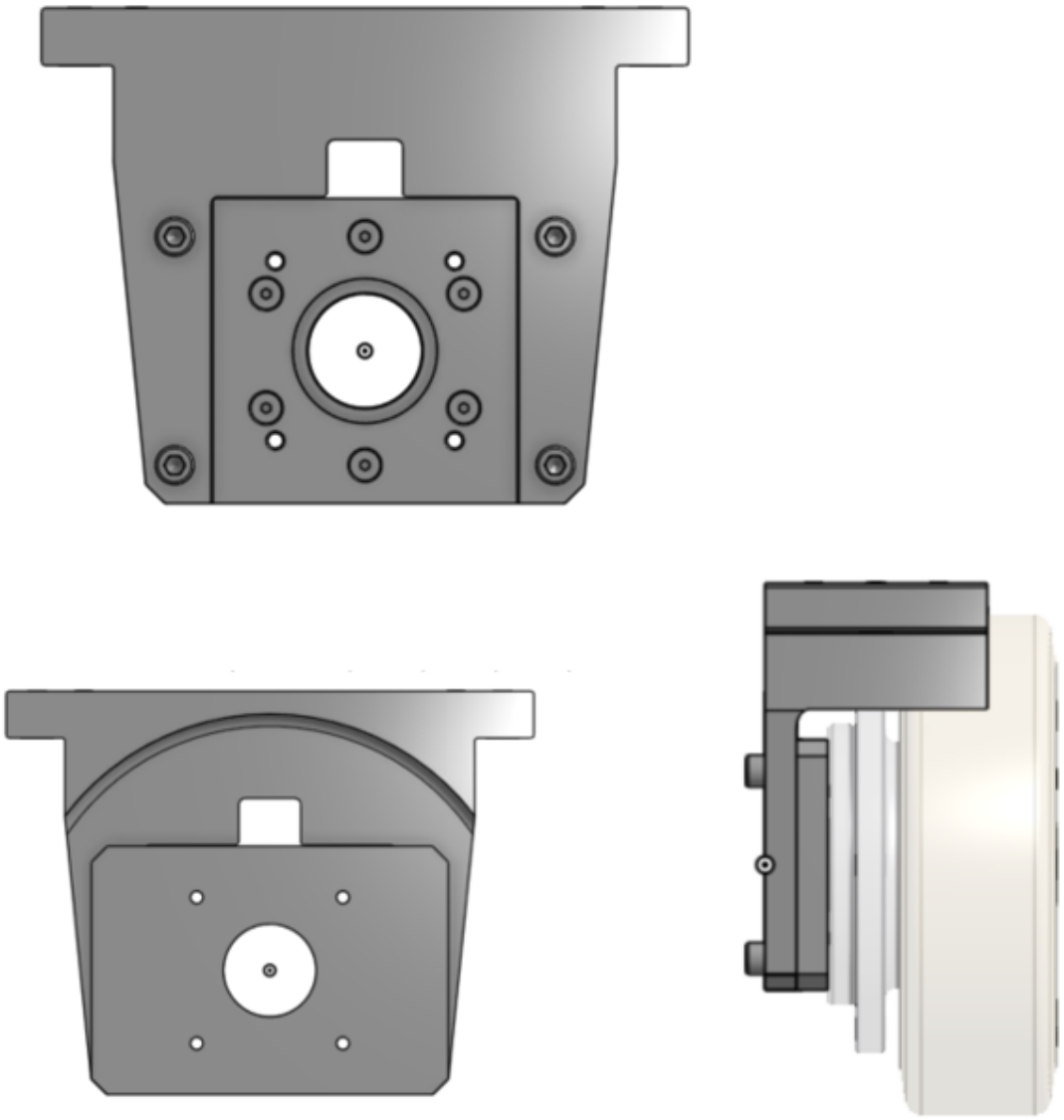


Figure 4.32: View on mounting bracket

4.1.23.3 How is the Wheel Drive Attached to the Chassis?

The mounting bracket has designated attachment points that align with corresponding points on the chassis. These points are typically threaded holes or slots that allow for bolts or screws to be used to secure the bracket to the chassis. Using appropriate bolts or screws, the mounting bracket is fastened to the chassis. It's important to use the correct type and size of fasteners to ensure a secure attachment. The bolts or screws are tightened to the specified torque to prevent loosening during operation.

Proper alignment of the wheel drive is crucial for optimal performance. The mounting bracket and attachment points are designed to ensure that the wheel is aligned correctly with the chassis, preventing any misalignment issues that could affect the movement and operation

of the AGV. Once the wheel drive is securely mounted, the necessary cabling for the motor and encoder is connected.

4.1.23.4 Benefits of the Mounting Design

The modular design of this mounting system offers unparalleled versatility, allowing for easy integration into various robotics platforms and making it suitable for a wide range of applications. The stable and secure attachment provided by the mounting bracket prevents any movement or vibrations that could affect performance, ensuring reliable operation. Additionally, the minimized cabling effort and straightforward attachment process make installation quick and efficient, significantly reducing downtime and maintenance costs. The system's ability to combine with different motors, brakes, and encoders allows for extensive customization based on specific application requirements, enhancing its adaptability and functionality.

4.1.24 The Caster Wheel and Its Components

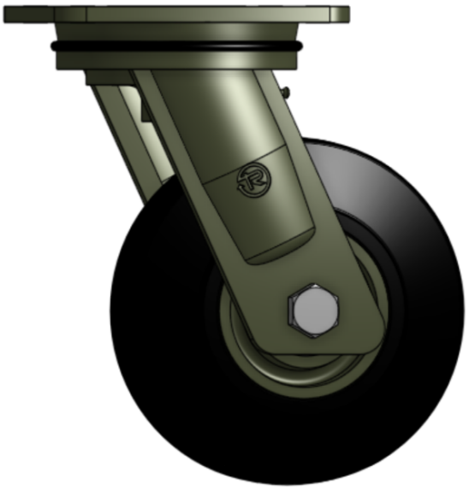


Figure 4.33: A caster wheel consists of several key components that work together to provide smooth and efficient mobility

4.1.24.1 Wheel:

This is the most visible part of the caster and is responsible for making contact with the ground. Wheels can be made from various materials such as rubber, polyurethane, nylon, phenolic, or cast iron, depending on the specific application and load requirements. Each material offers different benefits, like rubber providing a softer ride and cast iron offering high durability for heavy loads.

4.1.24.2 Mounting System

The mounting system connects the caster to the object it supports. There are two primary types: plate mounts and stem mounts. Plate mounts are used for heavier loads and provide a stable, secure attachment, while stem mounts are suitable for lighter applications where ease of installation is a priority.

4.1.24.3 Swivel Mechanism

This component allows the wheel to rotate 360 degrees, enhancing maneuverability. The swivel mechanism typically includes a raceway with ball bearings, which reduces friction and enables smooth, effortless swiveling. This is crucial for applications where tight turns and easy direction changes are necessary.

4.1.24.4 Rigid Frame

A fixed frame restricts the movement of the caster wheel to straight lines. This provides stability and control, making it ideal for applications where directional stability is essential, such as in straight-line travel paths.

4.1.24.5 Brake System

The brake system ensures that the caster stays stationary when needed. There are different types of brakes available, including wheel brakes that lock the wheel itself and swivel locks that prevent the swivel mechanism from turning. This added control is important for safety and precision in various applications.

4.1.24.6 Bearing

Bearings are installed inside the wheel hub or swivel mechanism to reduce friction and enable smooth rolling or swiveling. These components are critical for the efficient operation of the caster, as they ensure smooth, easy movement and reduce wear on the other parts.

4.1.24.7 Axle

The axle secures the wheel to the caster frame and ensures stability and alignment. It acts as a pivot point around which the wheel rotates and is usually made from durable materials to withstand the forces exerted during movement.

Together, these components create a caster wheel that provides reliable, smooth mobility for a wide range of applications, from furniture and medical equipment to industrial machin-

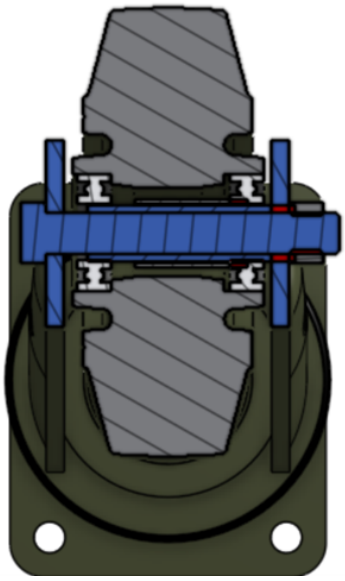


Figure 4.34: Caster wheel cross section view

ery and warehouse carts. The careful design and selection of each component ensure that the caster meets the specific needs of its intended use, providing both efficiency and durability.

4.1.24.8 What is a Deep Groove Ball Bearing?

A deep groove ball bearing is a widely popular type of rolling-element bearing, cherished for its versatility and efficiency. It consists of an inner ring, an outer ring, a cage, and a set of balls arranged in a circular pattern. The raceway grooves in both the inner and outer rings form a circular arc with a radius slightly larger than that of the balls, ensuring smooth and efficient rotation.

These bearings stand out for their high load capacity, supporting both radial and axial loads, making them suitable for a wide range of applications. They also have low frictional torque, enabling high rotational speeds and reduced energy consumption. Their simple and compact design allows for easy installation and minimal space requirements. Furthermore, they are optimized for low noise and vibration, making them ideal for applications where quiet operation is essential.

Available in single-row and double-row designs, as well as open, sealed, and shielded variants, deep groove ball bearings can be customized to meet specific application needs.

4.1.24.9 How Will the Inclination Be Handled?

Bearings with a spherical outer ring are designed to fit into housings with a concave bore, allowing them to compensate for any static misalignment of the shaft. This clever design ensures that even if the shaft isn't perfectly aligned, the bearing can still function

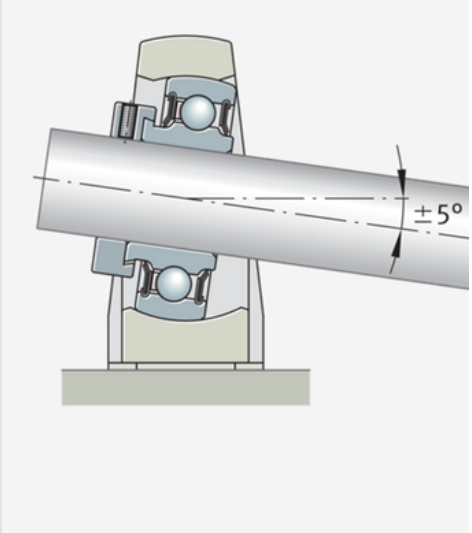


Figure 4.35: Bearing Inclination effect

smoothly.

However, there are limits to how much misalignment these bearings can handle. For maintenance-free housing units, the permissible angle of misalignment is $\pm 5°$. For housing units with a relubrication facility, the allowable misalignment angle is $\pm 2.5°$.

It's important to ensure that the center axes of the inner rings align on a straight line to achieve the best performance. This alignment ensures that the bearing operates efficiently and reduces the risk of premature wear or failure. This design is particularly useful in applications where maintaining perfect alignment is challenging, providing flexibility and reliability in various mechanical systems.

4.1.25 Load carrying capacity

4.1.25.1 Suitable for Very High Radial Loads

In radial insert ball bearings, the balls only make contact with the raceways at a single point. When the bearing is subjected to a purely radial load, these contact points lie at the center of the raceway. This means that the connection between the contact points passes directly through the radial plane, making the optimal load direction a purely radial one. Thanks to this design, radial insert ball bearings can support very high radial loads.

4.1.25.2 Larger Ball Sets Permit Higher Loads

The load-carrying capacity of a bearing depends on the bearing series and the size of the ball set within the reference bearings. For example, deep groove ball bearing series 60, which has a smaller bearing cross-section, cannot support loads as high as those of the standard series 62 of the same dimensions (relative to the bore diameter d), which features a larger

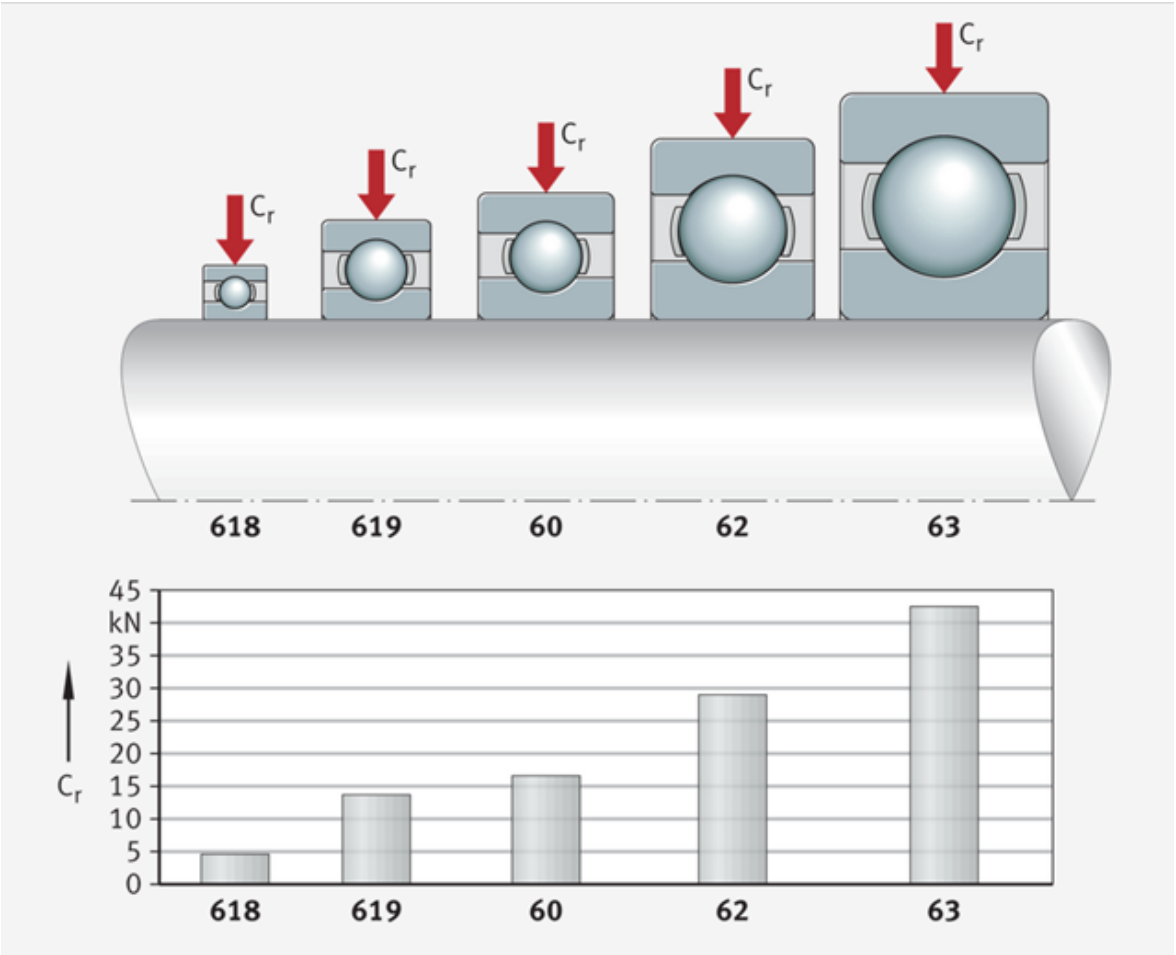


Figure 4.36: Load bearing chart

ball set. If even higher loads are required for the same bore diameter, the heavy bearing series 63, with the largest ball set, is the best choice. In summary, the design and size of the ball sets in radial insert ball bearings determine their ability to handle high radial loads, making them ideal for applications where such loads are present. This ability to support higher loads with larger ball sets ensures reliability and efficiency in a wide range of mechanical systems.

4.1.26 Engineering Procedure

4.1.26.1 Bearing selection

4.1.26.1.1 Inputs and Assumptions

1. Wheel radius (R): 110 mm
2. Load on the wheel (F): 981 N
3. Maximum speed (v): 1.25 m/s

4. Shaft material: Steel (e.g., AISI 1045)

    - Yield strength: σy=250 MPa

5. Bearing material: Chrome steel (e.g., AISI 52100)

6. Bearings catalog: We'll pick based on calculated load and RPM.

7. Factor of safety (FOS): 2.0 for shaft design.

4.1.26.1.2 Shaft Diameter Calculation

4.1.26.1.2.1 Angular Velocity

Convert the speed into angular velocity (ω) to determine the RPM.

$$\omega = \frac{v}{R} \tag{4.28}$$

Substitute values in eq. (4.28):

$$\omega = \frac{1.25}{0.06} = 20.83 \text{ rad/s}$$

Convert ω to RPM:

$$RPM = \omega \times 60/2\pi = 20.83 \times 60/2\pi \approx 199 \text{ RPM}$$

4.1.26.1.2.2 Bending Moment and Shaft Diameter

For a wheel shaft, the critical load is the bending moment due to the force F. Assuming a simple beam supported at the bearing points:

$$M = F \times L \tag{4.29}$$

Where L is the distance from the bearing to the load. Assume L=50 mm=0.05 m(minimum according to the ISO 3691-4:2020 standard):

$$M = 981 \times 0.05 = 49.05 \text{Nm}$$

Using the **maximum shear stress theory** for a solid circular shaft, the shaft diameter is calculated from:

$$d = (16M/\pi\tau max)^{1/3} \tag{4.30}$$

Where:

- τmax: Allowable shear stress = $\sigma y/2 \times FOS = 250/2 \times 2 = 62.5\,MPa = 62.5 \times 106\,Pa$

Substitute values in eq. (4.30):
$d = (16 \times 49.05/\pi \times 62.5 \times 106)^{1/3}$= 0.012mm

4.1.26.1.3 Bearing Selection

4.1.26.1.3.1 Load on the Bearing

The radial load on the bearing is equal to the load on the wheel:

$$Fr = 981\,\text{N}$$

4.1.26.1.3.2 Dynamic Load Rating

Using the bearing life equation:

$$C = Fr \times (L/1,000,000)^{\frac{1}{3}}$$

Where:

- L: Bearing life in revolutions. Assume L=10^6 revolutions.

$C = 981 \times (10^6/1,000,000)^{1/3} = 981\,N$
From a standard bearing catalog, a **deep groove ball bearing** with a dynamic load rating C>981 N and an inner diameter matching the shaft size (d=12 mm) is selected:

- **Bearing Type:** Deep groove ball bearing (e.g., 6201 series).
- **Verification :**

4.1.26.1.4 Material Properties of AISI 1045 Steel

Yield Strength (σy): Approximately 530 MPa
Ultimate Tensile Strength (σu): Approximately 625 MPa

4.1.26.1.5 Calculation

1. **Cross-Sectional Area (A)**:

$$A = \pi \left(\frac{d}{2}\right)^2 = \pi \left(\frac{12\,\text{mm}}{2}\right)^2 \approx 113.1\,\text{mm}^2$$

2. **Stress (σ)**:

A shaft of 12mm diameter made of AISI 1045 steel can safely bear a load of 981N, as the induced stress is well within the material's yield and ultimate tensile strengths.

$$\sigma = \frac{F}{A} = \frac{981 \text{ N}}{113.1 \text{ mm}^2} \approx 8.67 \text{ MPa}$$

4.1.26.1.5.1 3.3 Number of Balls in the Bearing

To determine the diameter of the balls in the bearing when we are supposed to have 9 balls(i tried all the number less than 9 , they were not compatible with the standard), we can use the following formula:

$$\text{z} = \frac{\pi(D+d)}{2d_b} \tag{4.31}$$

where: z, the number of balls; D, the outer diameter of the bearing; d, the inner diameter of the bearing; and db, the diameter of each ball. For this specific case, the given values are as follows: $z = 9$, $D = 32\,\text{mm}$, and $d = 12\,\text{mm}$.

We need to solve for db:

$$\begin{aligned} z &= \frac{\pi(D+d)}{2db} \\ 9 &= \frac{\pi(32+12)}{2db} \\ 9 &= \frac{\pi \cdot 44}{2db} \\ db &= \frac{\pi \cdot 44}{2 \cdot 9} \\ db &\approx 7.68\,\text{mm} \end{aligned}$$

For a 6201 bearing:

- Number of balls: Typically 7–9 balls.
- Ball diameter: beyond 4.5 mm.

4.1.26.1.6 Material Selection

4.1.26.1.6.1 Shaft Material: AISI 1045 Steel

- Reason: High strength, good machinability, and readily available.

4.1.26.1.6.2 Bearing Material: AISI 52100 Chrome Steel

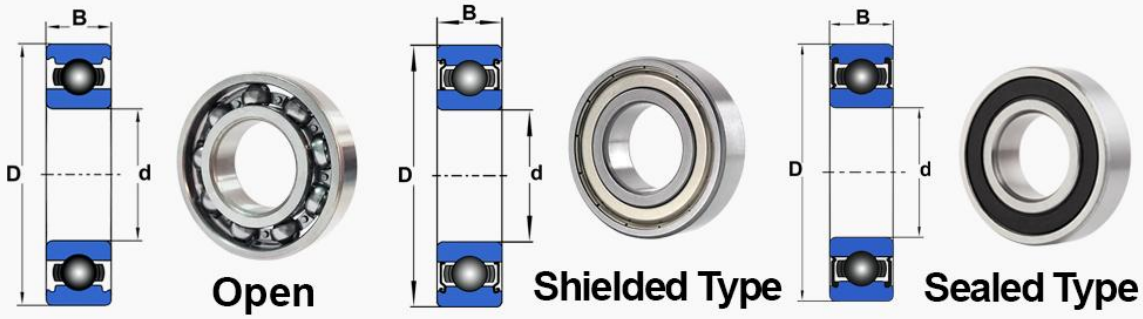
- Reason: High hardness, wear resistance, and durability under dynamic loads.

4.1.26.1.7 Compatibility Check

A deep groove ball bearing that meets these specifications is the **NSK 6201** bearing. Here are the details:

- **Inner Diameter (ID)**: 12 mm
- **Outer Diameter (OD)**: 32 mm
- **Width (W)**: 10 mm
- **Material**: AISI 52100 Chrome Steel
- **Load Capacity**: Suitable for the given load and speed requirements

NSK Deep Groove Ball Bearing



Boundary Dimensions (mm)				Basic Load Ratings (N)		{kgf}		Factor	Limiting Speeds (rpm) Grease		Oil	Bearing Numbers			
d	D	B	r min	C_r	C_{0r}	C_r	C_{0r}	f_0	Open Z · ZZ V · VV	DU DDU	Open Z	Open	Shielded		Sealed
10	19	5	0.3	1 720	840	175	86	14.8	34 000	24 000	40 000	**6800**	**ZZ**	**VV**	**DD**
	22	6	0.3	2 700	1 270	275	129	14.0	32 000	22 000	38 000	**6900**	**ZZ**	**VV**	**DD**
	26	8	0.3	4 550	1 970	465	201	12.4	30 000	22 000	36 000	**6000**	**ZZ**	**VV**	**DDU**
	30	9	0.6	5 100	2 390	520	244	13.2	24 000	18 000	30 000	**6200**	**ZZ**	**VV**	**DDU**
	35	11	0.6	8 100	3 450	825	350	11.2	22 000	17 000	26 000	**6300**	**ZZ**	**VV**	**DDU**
12	21	5	0.3	1 920	1 040	195	106	15.3	32 000	20 000	38 000	**6801**	**ZZ**	**VV**	**DD**
	24	6	0.3	2 890	1 460	295	149	14.5	30 000	20 000	36 000	**6901**	**ZZ**	**VV**	**DD**
	28	7	0.3	5 100	2 370	520	241	13.0	28 000	—	32 000	**16001**	—	—	—
	28	8	0.3	5 100	2 370	520	241	13.0	28 000	18 000	32 000	**6001**	**ZZ**	**VV**	**DDU**
	32	10	0.6	6 800	3 050	695	310	12.3	22 000	17 000	28 000	**6201**	**ZZ**	**VV**	**DDU**
	37	12	1	9 700	4 200	990	425	11.1	20 000	16 000	24 000	**6301**	**ZZ**	**VV**	**DDU**
15	24	5	0.3	2 070	1 260	212	128	15.8	28 000	17 000	34 000	**6802**	**ZZ**	**VV**	**DD**
	28	7	0.3	4 350	2 260	440	230	14.3	26 000	17 000	30 000	**6902**	**ZZ**	**VV**	**DD**
	32	8	0.3	5 600	2 830	570	289	13.9	24 000	—	28 000	**16002**	—	—	—
	32	9	0.3	5 600	2 830	570	289	13.9	24 000	15 000	28 000	**6002**	**ZZ**	**VV**	**DDU**
	35	11	0.6	7 650	3 750	780	380	13.2	20 000	14 000	24 000	**6202**	**ZZ**	**VV**	**DDU**
	42	13	1	11 400	5 450	1 170	555	12.3	17 000	13 000	20 000	**6302**	**ZZ**	**VV**	**DDU**
17	26	5	0.3	2 630	1 570	268	160	15.7	26 000	15 000	30 000	**6803**	**ZZ**	**VV**	**DD**
	30	7	0.3	4 600	2 550	470	260	14.7	24 000	15 000	28 000	**6903**	**ZZ**	**VV**	**DDU**
	35	8	0.3	6 000	3 250	610	330	14.4	22 000	—	26 000	**16003**	—	—	—
	35	10	0.3	6 000	3 250	610	330	14.4	22 000	13 000	26 000	**6003**	**ZZ**	**VV**	**DDU**
	40	12	0.6	9 550	4 800	975	490	13.2	17 000	12 000	20 000	**6203**	**ZZ**	**VV**	**DDU**
	47	14	1	13 600	6 650	1 390	675	12.4	15 000	11 000	18 000	**6303**	**ZZ**	**VV**	**DDU**
20	32	7	0.3	4 000	2 470	410	252	15.5	22 000	13 000	26 000	**6804**	**ZZ**	**VV**	**DD**
	37	9	0.3	6 400	3 700	650	375	14.7	19 000	12 000	22 000	**6904**	**ZZ**	**VV**	**DDU**
	42	8	0.3	7 900	4 450	810	455	14.5	18 000	—	20 000	**16004**	—	—	—
	42	12	0.6	9 400	5 000	955	510	13.8	18 000	11 000	20 000	**6004**	**ZZ**	**VV**	**DDU**
	47	14	1	12 800	6 600	1 300	670	13.1	15 000	11 000	18 000	**6204**	**ZZ**	**VV**	**DDU**
	52	15	1.1	15 900	7 900	1 620	805	12.4	14 000	10 000	17 000	**6304**	**ZZ**	**VV**	**DDU**
22	44	12	0.6	9 400	5 050	960	515	14.0	17 000	11 000	20 000	**60/22**	**ZZ**	**VV**	**DDU**
	50	14	1	12 900	6 800	1 320	695	13.5	14 000	9 500	16 000	**62/22**	**ZZ**	**VV**	**DDU**
	56	16	1.1	18 400	9 250	1 870	940	12.4	13 000	9 500	16 000	**63/22**	**ZZ**	**VV**	**DDU**

Figure 4.37: Bearing boundary dimensions

4.1.26.2 Torque calculation

4.1.26.2.1 Given Data:

- Gross Vehicle Weight: 300 kg
- Weight per Drive Wheel: 100 kg
- Maximum Incline Angle: 8°
- Maximum Velocity: 1.5 m/s
- Surface: Concrete
- Type of Bearing: Deep Groove Ball Bearing (NSK 6201)
- Wheel Type: Rubber Tires
- Drive Wheel Diameter: 95 mm (0.095 m)

4.1.26.2.2 Load Analysis:

1. **Load Calculation**:

   $F = W = 100\,kg \times 9.81\,m/s2 = 981\,N$

2. **Wheel Rotational Speed**:

   $r = 0.22\,m/2 = 0.11\,m$

   $Circumference = \pi \times d = \pi \times 0.22\,m = 0.6909\,m$

   Wheel Rotational Speed=Linear Speed

   $Circumference = 1.25\,m/s \;/\; 0.6909\,m \approx 1.81\,r/s \approx 108.6\,RPM$

3. **Required Torque**:

   $T = F \times r = 981\,N \times 0.11\,m = 107.91\,Nm$

   Considering efficiency and safety:

   $Efficiency(85\%), Tmotor = Tt/Eff. = 107.91\,Nm/0.85 = 126.95\,Nm$

   (without bearing)

4. **Torque Required for Acceleration**: Assuming α=1m/sˆ2:

   Require Tractive effort (Ft)

   $Ft = mt\ x\ \alpha =\ 300\ kg\ x\ 1\ m/s^2 = 300N$

Force must be provided by frictional force at the wheel

$T = Ft\ x\ r = 300N\ x\ 0.11m = 33Nm$

5. **Effect of Inclination**:

   $\theta = 3^\circ = 0.0524\,rad$

   $Gravity\ Component = 300 \times 9.81 \times sin(0.0524) \approx 154.11N$

   $Tincline = \ Fgravity\ x\ r = 154.11N\ x\ 0.11m = 16.9521Nm$

6. **Torque Due to Friction**: Assuming μ=0.002(coef. friction):16.9521 Nm

$Ffriction = \mu\ x\ R\ =\ 0.002\ x\ 981N = 1.962N$ (R: radial load)
$Torque\ Due\ to\ Friction = Ff \times r = 1.962N \times 0.006m = 0.011772Nm$ (r=bore radius)

Total Torque Required:
$Ttotal = T1 + Tfriction = 33Nm + 0.011772Nm = 33.211772Nm$
Verification:

- The NSK 6201 bearing has a basic dynamic load rating of 7.28 kN, which is sufficient to handle the load of 981 N.
- The calculated torque values and rotational speed are within the bearing's specifications.

4.1.26.3 Results

The outcomes of the computations and validation procedures are as follows:

4.1.26.3.1 Shaft Measurements

12 mm is the calculated shaft diameter.
Approximately 8.67 MPa of induced stress is much less than the material's yield strength of 250 MPa.

4.1.26.3.2 Bearing Details:

NSK 6201 deep groove ball bearing model.
Dimensions:
- 12 mm in diameter within.
- 32 mm in diameter outside.
- Capacity to load: Equipped to manage loads that are above the designated 981 N.
- Ball diameter: 7.68 mm, which meets catalog requirements for nine balls.

4.1.26.3.3 Observance of Standards

By following ISO 3691-4:2020 guidelines, the design guarantees load-handling systems' dependability and safety. For operating performance, the shaft and bearing dimensions meet the minimal requirements.

4.1.26.3.4 Total Torque Required:

The dynamic load needed is 28.2Nm

4.2 ELECTRICAL DESIGN

4.2.1 Electrical System Design

At the core of an AGV's electrical system are power management, motor control, and communication networks, all of which must be carefully designed to meet performance and safety requirements. The power system typically consists of a rechargeable battery, often lithium-ion or lead-acid, along with a Battery Management System (BMS) to monitor voltage, current, and temperature for optimal energy efficiency and longevity. Motor controllers regulate the movement of the drive and steering motors, ensuring smooth acceleration, precise navigation, and effective braking. Additionally, AGVs rely on a network of embedded controllers, sensors, and communication modules to process data in real time, enabling obstacle detection, localization, and coordination a central control system.

4.2.2 Power Supply and Distribution System

At the heart of the AGV's electrical design lies the power supply and distribution system, which serves as the backbone for all onboard functionalities. This system is responsible for storing, managing, and delivering electrical energy to various components, including motors, sensors, control systems, and communication modules. Given that AGVs must operate continuously for extended periods, the selection of battery technology, capacity, and management strategies plays a critical role in determining performance, efficiency, and reliability. The AGV must be able to complete all assigned tasks without running out of power, making energy efficiency and power optimization essential in the design.

To ensure safe and efficient energy usage, the AGV's battery system must integrate a Battery Management System (BMS). This system continuously monitors key parameters such as voltage, current, temperature, and state of charge, ensuring that the battery operates within safe limits while optimizing power delivery.

Equally important is the power distribution architecture, which ensures reliable and efficient energy delivery to all AGV subsystems. This involves designing robust wiring harnesses, fuses, circuit breakers, and connectors capable of handling dynamic load requirements during various operational phases.

The following section explores the key components of the AGV's power supply and distribution system, their functions, and how they contribute to the overall efficiency and reliability of the vehicle.

4.2.2.0.1 Power distribution architecture (System architecture)

A well-structured power distribution network minimizes voltage drops and energy losses, directly contributing to longer operating times and reduced downtime for recharging. Furthermore, if the expected task duration exceeds the battery capacity, solutions such as hot-swappable batteries or rapid charging stations should be considered to maintain uninterrupted operations.

Figure 4.38 illustrates an efficient power distribution system [17]for an AGV. The architecture includes key components such as the battery, power management system, motor controllers, sensors, microcontroller, and embedded computer. The power supply must ensure reliable and continuous energy delivery while optimizing efficiency and minimizing losses. The power requirements of an AGV are primarily determined by the embedded-PC, microcontroller, sensors, and motors. The microcontroller is responsible for low-level control, including direct motor actuation and sensor data acquisition. Additionally, it provides a programming interface for the embedded-PC, which has higher computational capabilities and is used for high-level tasks, such as motion planning and coordination.

Since the battery supplies direct current (DC), different power conversion techniques are required to meet the voltage demands of various components. The high-voltage DC motors require DC-DC boosters to step up the supplied voltage to the necessary operating level. Meanwhile, low-voltage components such as sensors, the microcontroller, and embedded-PC rely on DC voltage regulators to ensure stable power delivery. This architecture enables precise control over motor speed through pulse width modulation (PWM), allowing for efficient and adaptive movement of the AGV.

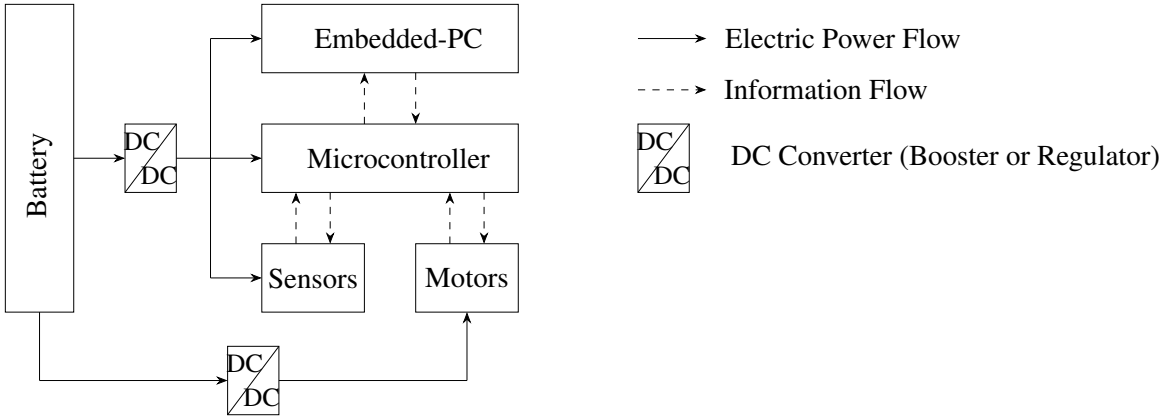


Figure 4.38: Typical architecture of an automated guided vehicle in terms of the electrical power distribution.

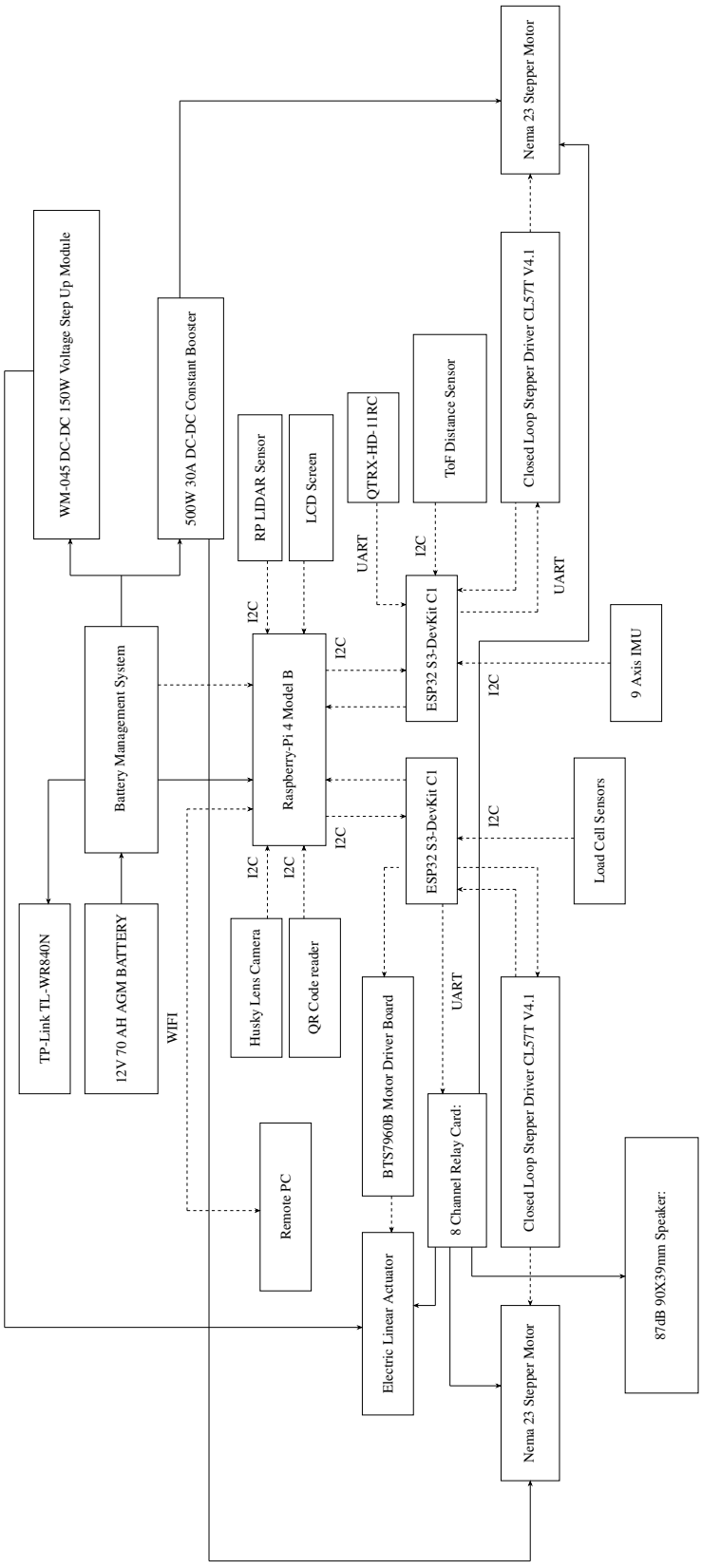


Figure 4.39: Actual system architecture

The CIU_Fox robot features a sophisticated electrical and control system integrating mul-

tiple embedded components, sensors, and power management units to ensure efficient operation. At the core of its control system are two ESP32 S3-DevKit C1 microcontrollers, responsible for handling various sensor inputs and executing real-time processing. These microcontrollers communicate with a Raspberry Pi 4 Model B, which serves as the main computational unit, facilitating high-level processing and decision-making. The Raspberry Pi is interconnected with several peripherals through I2C and UART communication protocols, ensuring seamless data transmission between components such as the Husky Lens camera, QR code reader, and RP LIDAR sensor, which contribute to the robot's vision and navigation capabilities. Additionally, an LCD screen displays real-time feedback (Sensors output,error message,...), while a 9-axis IMU enhances motion sensing for precise movement adjustments.

For actuation, the robot relies on Nema 23 stepper motors, which are powered and controlled by Closed Loop Stepper Driver CL57T V4.1 modules, ensuring precise rotational movements. These motors work in conjunction with an electric linear actuator (Lifting mechanism) controlled via a BTS7960B motor driver board. To manage additional mechanical operations and ensure safety, an 8-channel relay card is integrated into the system. This relay card not only coordinates multiple control signals for precise operation but also serves as a critical safety feature by isolating power components, such as motors and brakes, from the main circuit in case of an emergency. This dual functionality ensures both reliable control and robust fail-safe protection during operation. The CIU_Fox robot is also equipped with load cell sensors, allowing it to measure forces applied to its lifting mechanism.

Distance sensing and obstacle avoidance are managed by a Time-of-Flight (ToF) sensor and a LiDAR system, which together provide precise environmental awareness for navigation and object detection. Complementing these systems, the QTRX-HD-11RC modules work alongside the camera to deliver robust line-following capabilities, ensuring accurate path tracking and reliable navigation in structured environments.

A 12V 70AH AGM battery serves as the primary power source, delivering energy to various subsystems. A Battery Management System (BMS) regulates power flow, preventing overcharging and ensuring optimal energy usage. Voltage regulation is achieved using a combination of modules, including a WM-045 DC-DC 150W step-up converter, a 500W 30A DC-DC constant booster, and LM2576 buck converters for 3.3V, 5V, and 9V outputs, ensuring stable and precise voltage levels for all components. To enhance system functionality, an 87dB speaker is integrated, providing auditory alerts or notifications as needed. Wireless connectivity is established through a TP-Link TL-WR840N router, enabling remote access and control via a connected PC, which facilitates real-time monitoring and decision-making.

4.2.2.0.2 Battery Management System (BMS)

The **BMS** is integrated to monitor and manage the battery's health, ensuring safe and reliable operation. It oversees critical parameters such as voltage, current, and temperature, preventing issues like overcharging, over-discharging, and overheating. Having a BMS typically optimizes battery performance, extending battery life by up to 30% and supporting over 3500 charge-discharge cycles at 90% Depth of Discharge (DOD). It also ensures safe operations with features like overcurrent protection, overvoltage protection, and temperature monitoring. Additionally, the BMS in this system includes a charging system that complies with the competition specifications, which supports an **AGM Battery** with a nominal voltage of **12VDC** and a charging current of **10A**.

4.2.2.1 Battery System

Battery selection is a key factor in the efficiency and reliability of an AGV system. Among the available options, Absorbent Glass Mat (AGM) and Gel (GEL) batteries, both types of Sealed Lead-Acid (SLA) or Valve-Regulated Lead-Acid (VRLA) batteries, stand out for their maintenance-free, leak-proof design and low self-discharge rate. While both are deep-cycle batteries capable of discharging up to 80% of their capacity, GEL batteries emphasize durability and stable power delivery, whereas AGM batteries support higher discharge rates and slightly faster charging. However, neither is well-suited for opportunity charging, as frequent partial recharges reduce their lifespan.

For AGVs operating in controlled, single-shift environments, GEL batteries provide a strong balance between safety, longevity, and consistent performance, making them an optimal choice. They typically last 8–16 hours per charge, depending on the AGV type, and require a full recharge when reaching 40–50% depth of discharge (DOD). While their charging time is slower (e.g., a 100Ah battery takes about 3 hours at 0.3C), their deep-cycle capability ensures steady, long-term operation, minimizing the need for frequent replacements.

Considering the AGV's operational demands and the competition's specified battery requirements, AGM batteries are selected for this application. The competition specifies the following parameters: *Battery Type: AGM Battery, Battery Voltage (Nominal): 12VDC, Battery Charging Current: 10A*.

Figure 4.40: AGM battery

4.2.2.2 Voltage Regulation

The AGV's electrical system contains various components operating at different voltage levels, making voltage regulation and distribution critical for stable operation. The majority of sensors operate at 3.3V to 5V, while components like the Raspberry Pi and ESP32 require 5V and others like the Wi-Fi router operates at 9V. To accommodate these diverse requirements, multiple voltage regulators are incorporated to step down the 12V battery supply to the appropriate levels, ensuring consistent performance across all subsystems.

The **LM2596** in Figure 4.43 voltage regulator will be used to step down the 12V battery supply to the required voltage levels for various components. Different versions of the LM2596 are available, including fixed output versions (9V, 5V, and 3.3V) as well as an adjustable version, allowing flexibility to meet the specific voltage requirements of each subsystem. For example, the 5V version will power the Raspberry Pi and ESP32, while the 3.3V version will support sensors operating at lower voltages. In cases where multiple components require the same voltage and draw significant current, multiple LM2596 regulators are placed in parallel to increase the current supply capacity(Table4.12).

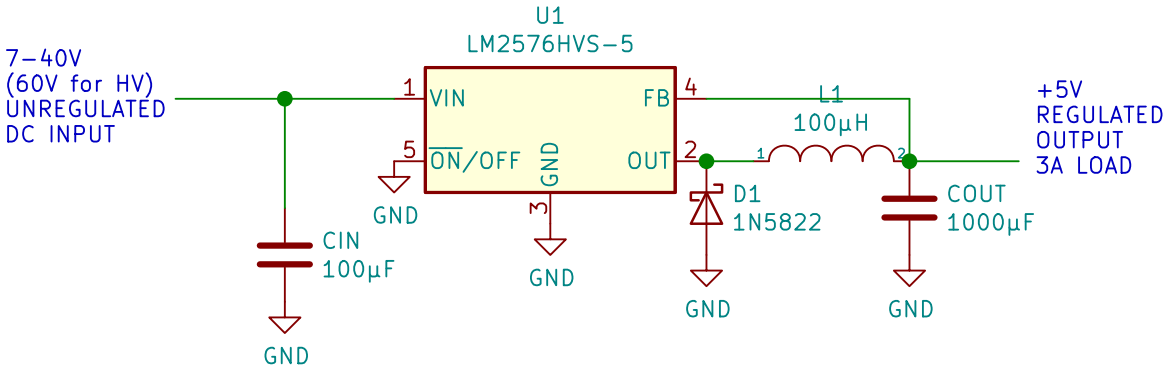


Figure 4.41: Fixed Output Voltage Version Typical Application Diagram

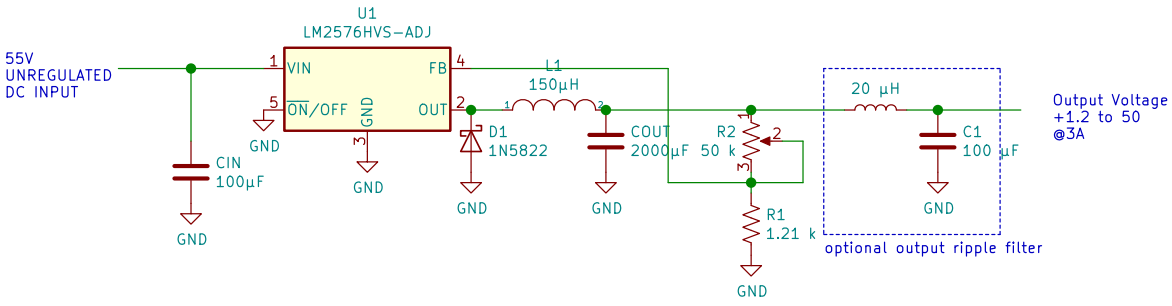


Figure 4.42: 1.2-V to 55-V Adjustable 3-A Power Supply With Low Output Ripple

Figure 4.43: LM2596 Voltage Regulator

Parameter	Value
Input Voltage Range	Up to 40V
Output Current	Up to 3A
Switching Frequency	150 kHz
Output Voltage Options	Fixed: 3.3V, 5V, 9V Adjustable: 1.2V--37V
Efficiency	90%
Protection Features	Thermal shutdown, current limiting
Package Type	TO-220, TO-263
Operating Temperature	-40°C to +125°C

Table 4.12: LM2596 Voltage Regulator Specifications

Furthermore, subsystems, such as the traction unit (composed of two stepper motors) and the lifting system (powered by linear actuators), require voltages higher than the battery's output. To address this, boost converters(fig. 4.44 and fig. 4.45) are used to step up the voltage to the necessary levels.

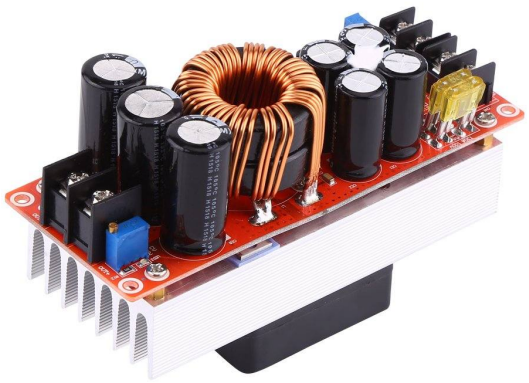


Figure 4.44: 1500W 30A DC-DC Constant Current Boost Converter Step-up Power Supply Module 10-60V to 12-90V

Features:	Flexible DC input voltage range from 10V to 60V; Adjustable output voltage from 12V to 90V for versatile applications; Maximum output current of 20A for reliable power supply; Built with high-power 100V/210A low resistance MOS for efficiency; Reverse input protection with MOS for added safety; Low voltage protection to prevent damage from over-discharge; Thickened heat sink and intelligent cooling fan for optimal heat dissipation.
Specifications:	Power: 1500W; Current: 30A; Input Voltage Range: 10V - 60V; Output Voltage Range: 12V - 90V; Maximum Output Current: 20A.
Package Includes:	1 × Boost Converter Step-up Power Supply Module.

Table 4.13: DC-DC Constant Current Boost Converter Specifications and Features

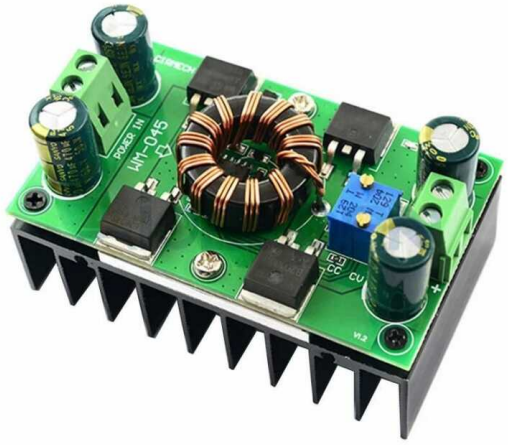


Figure 4.45: WM-045 DC-DC 150W Voltage Step Up and Step Down Regulator Module

Features:	Input Voltage: 5Vdc 30Vdc; Buck-Boost (automatic boost/lower); Output Voltage: 1.25Vdc -- 30Vdc (Adjustable); Output Current: 10A (MAX).
Specifications:	Output Power: 150W; Conversion Efficiency: 90% Max; Ripple and Noise: 200mVp-p; No-Load Current: 6mA typical.
Performance Metrics:	Voltage Regulation: ± 0.5%; Load Regulation: ± 0.5%; Dynamic Response Rate: 300uS.

Table 4.14: Buck-Boost Converter Specifications

The power distribution circuits are designed to efficiently deliver power to all subsystems, including motors, sensors, controllers, and communication modules, while minimizing power losses and ensuring reliable operation. Protection circuits and fuses are also integrated to safeguard sensitive components from voltage spikes, short circuits, and other electrical faults, enhancing the overall safety and durability of the AGV.

4.2.3 Motor Control System

The performance and reliability of automated guided vehicles (AGVs) are deeply rooted in the precision and efficiency of their motor control systems [18]. At the heart of these systems lies the electrical design, which governs the interaction between power supply, control devices, and the physical dynamics of the motors. Direct current (DC) motors [19], Stepper motors [20] and Servo motors [21], widely used in AGVs due to their high torque, excellent speed regulation, and robust performance, require sophisticated electrical architectures to ensure optimal operation.

From an electrical perspective, motor control involves not only the precise regulation of voltage and current but also the implementation of efficient drivers and motor controllers, which are critical for achieving smooth acceleration, stable speed control, and efficient energy utilization. Furthermore, the motor control system must address challenges such as minimizing power losses, managing heat dissipation, and ensuring reliable operation under varying load conditions.

A motor control system comprises two fundamental components: the motors themselves and the motor drivers or controllers. The motor alone cannot achieve optimal operation without the integration of a sophisticated motor driver or controller. The motor driver acts as the intermediary between the power supply and the motor, regulating voltage and current to ensure smooth and efficient operation. It interprets control signals and translates them into precise commands.

4.2.3.1 Motors and actuators

The choice of motor depends on several factors, including torque requirements, speed, wheel size, load capacity, operating voltage, and environmental conditions. Below, we discuss the key considerations for motor selection in AGV applications.

4.2.3.1.0.1 Torque and Speed Requirements

The motor must generate sufficient torque to move the AGV and its payload efficiently, overcoming rolling resistance and any potential obstacles. Simultaneously, it should achieve

the desired speed without compromising torque. For instance, AGVs operating in environments with frequent stops and starts may require motors with high torque at low speeds, while those used in long-distance transportation may prioritize higher speed capabilities.

4.2.3.1.0.2 Wheel Size and Load Capacity

The size and weight of the AGV wheels directly influence motor selection. Larger wheels and heavier payloads demand motors with higher torque and power ratings. Additionally, the motor must be capable of handling the AGV's maximum load capacity without overheating or losing efficiency.

4.2.3.1.0.3 Operating Voltage

The motor's operating voltage must align with the AGV's power supply system. Common voltage options include 12V, 24V, and 48V. It is essential to choose a voltage that is both compatible with the AGV's design and readily available for the intended application.

4.2.3.1.0.4 Motor Types

The two most common motor types for AGVs are DC motors and brushless DC (BLDC) motors. DC motors are cost-effective and straightforward, making them suitable for simpler applications. On the other hand, BLDC motors offer superior efficiency, longer lifespan, and smoother operation, making them ideal for more demanding AGV tasks. The choice between these motor types depends on the specific requirements of the AGV, such as precision, durability, and control complexity.

4.2.3.1.0.5 Control System Compatibility

The selected motor must integrate seamlessly with the AGV's control system. This includes compatibility with communication protocols and interfaces, such as PWM (Pulse Width Modulation) for speed control or CAN bus for advanced communication. Ensuring compatibility is crucial for achieving precise control and coordination with other AGV subsystems.

4.2.3.1.0.6 Encoders and Feedback Mechanisms

For AGVs requiring high precision and accurate path correction, motors equipped with encoders or feedback mechanisms are highly recommended. Encoders provide real-time data on the motor's position and speed, enabling closed-loop control and improved navigation

accuracy. This feature is particularly important for AGVs operating in dynamic or complex environments.

4.2.3.1.0.7 Environmental Considerations

The operating environment plays a significant role in motor selection. AGVs operating in harsh conditions, such as dusty or moist environments, require motors with appropriate protection ratings (e.g., IP65 or higher). Additionally, motors should be selected based on their ability to withstand temperature variations and mechanical stress.

4.2.3.1.0.8 Energy Efficiency

Energy efficiency is a critical factor, especially for AGVs that operate for extended periods or rely on battery power. Motors with high efficiency reduce power consumption, extend battery life, and lower operational costs. BLDC motors, for example, are known for their energy-efficient performance compared to traditional DC motors.

4.2.3.1.0.9 Cost and Availability

While selecting a motor, it is essential to balance performance requirements with budget constraints. Consider not only the initial cost of the motor but also its availability, maintenance requirements, and long-term reliability. Consulting with manufacturers and industry experts can help identify cost-effective solutions without compromising performance.

4.2.3.1.1 Traction unit motors and linear actuator

4.2.3.1.1.1 P Series Nema 23 Closed Loop Stepper Motor with Electromagnetic Brake (2Nm):

The traction system of the automated guided vehicle (AGV) is driven by two NEMA 23 P-Series stepper motors shown Figure 4.46, each delivering a holding torque of 2.0 Nm to ensure reliable and precise motion. These motors are selected for their high torque-to-size ratio, making them well-suited for the dynamic demands of AGV operations.

Both motors are equipped with incremental encoders, which provide real-time feedback on motor speed and position. This closed-loop system enhances the AGV's ability to maintain smooth and synchronized motion, ensuring stability and precision during navigation.

Additionally, each motor is integrated with an electromagnetic brake, enabling rapid deceleration and enhancing safety by halting motion instantly when required.

Figure 4.46: Nema 23 stepper motor 2 Nm torque

Electrical Specification	
Manufacturer Part Number	23E1KBK20-20
Number of Phases	2
Step Angle	1.8deg
Holding Torque	2Nm (283.28oz.in)
Rated Current/Phase	5.0A
Phase Resistance	0.4ohms
Inductance	1.8mH ± 20% (1KHz)
Insulation Class	B 130°C [266°F]
Physical Specification	
Frame Size	57 x 57mm
Body Length	76.5mm
Shaft Diameter	8mm
Shaft Length	21mm
D-Cut Shaft Length	15mm
Lead Length	270mm
Weight	1.06kg
Encoder Specification	
Output Circuit Type	Differential type
Encoder Type	Incremental
Output Signal Channels	2 channels
Supply Voltage Min	5V
Supply Voltage Max	5V
Output High Voltage	<4V
Output Low Voltage	<1V
Brake Connections	
Red–24V+	Red--24V-
Brake Torque	2.0Nm

Table 4.15: Stepper Motor Specifications (Model: 23E1KBK20-20)

paragraphDC 12V Electric Linear Actuator (Maximum Force 6000N):

The lifting mechanism of the automated guided vehicle (AGV) utilizes a high-performance linear actuator to achieve precise vertical motion. The actuator includes an overload protection feature, preventing excessive force application and minimizing mechanical wear.

Figure 4.47: Linear actuator with a maximum stoke of 6000N

Model	JS-TGZ-U3
Material	Metal
Voltage	DC 12V
Maximum Push/Pull Force	Approx. 6000N/6000N
Stroke	450mm
No-Load Speed	Maximum 5mm/s
Rated Load Rate	5mm/s (600kg)
Environment Temperature	-26°C to +65°C
Standard Protection Level	IP54
Built-in Stroke Switch	Yes
Color	Silver

Table 4.16: Specifications of the linear actuator model JS-TGZ-U3

4.2.3.1.2 Motor Drivers and Controllers

4.2.3.1.2.1 Closed Loop Stepper Driver V4.1 CL57T:

The **Closed Loop Stepper Driver V4.1 CL57T** has been selected to complement the **Nema 23 Closed Loop Stepper Motor**, ensuring precise and reliable control over the AGV's propulsion and mechanical functions.

From an electrical perspective, the CL57T driver leverages advanced closed-loop technology to guarantee accurate motor positioning, even in dynamic environments or when

encountering external resistance. Fully compatible with software tools like **CLseries** and **Motion Studio**, the driver enables fine-tuning of critical motor parameters, optimizing performance for specific operational requirements.

Additionally, its extensive diagnostic and monitoring ports provide real-time tracking of key electrical and operational metrics, such as **temperature**, **current**, **torque**, **RPM**, and overall motor state during operation. These capabilities not only enhance operational visibility but also streamline troubleshooting and diagnostics, minimizing downtime and ensuring swift resolution of issues.

Key Features	
RS232 debugging interface Do not need a high torque margin Broader operating speed range Reduced motor heating and more efficient Smooth motion and super-low motor noise 5V/24V logic voltage selector, default setting 24V Closed-loop, eliminates loss of synchronization Protections for over-voltage,over-current,and position following error By default, supports an encoder with a resolution of 1000PPR; customizable between 0-5000PPR	
Electrical Specifications	
Output Peak Current	0 8A
Input Voltage	+24 48VDC (Typical 36VDC)
Logic Signal Current	7 16mA (Typical 10mA)
Pulse Input Frequency	0 200kHz
Isolation Resistance	500Mohm
Operating Environment and Other Specifications (Tj = 25°C/77°F)	
Cooling	Natural Cooling or Forced Cooling
Environment	Avoid dust, oil mist, and corrosive gases
Ambient Temperature	0°C - 65°C
Humidity	40%RH - 90%RH
Operating Temperature	0°C - 50°C
Vibration	10-50Hz / 0.15mm
Storage Temperature	-20°C - 65°C
Weight	Approx. 280g (9.9oz)

Table 4.17: Specifications of the Closed-Loop Stepper Driver

Figure 4.48: Closed Loop Stepper Driver CL57T V4.1

4.2.3.1.2.2 BTS7960B 40 Amp Motor Driver Board

The BTS7960B 40A Motor Driver Board is selected for its capability to control high-power motors, making it ideal for the AGV's linear actuator. Supporting currents up to 40A, it ensures reliable performance in demanding applications.

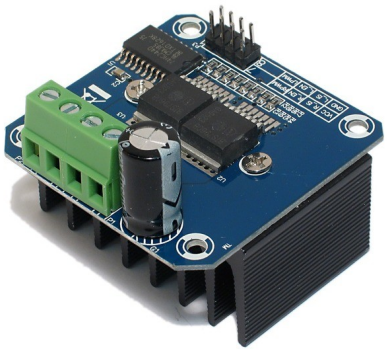


Figure 4.49: DC-MOTOR DRIVER MODULE 24V 43A (2x BTS7960B)

Features	
Double BTS7960 large current (43A) H-bridge driver 5V isolation with MCU, effectively protecting the MCU 5V power indicator on board Voltage indication of motor driver output end Can solder heat sink for improved thermal management Requires only four lines from MCU to driver module (GND, 5V, PWM1, PWM2) Isolation chip 5V power supply (can share with MCU 5V) Supports motor forward and reverse control; two PWM input frequencies up to 25kHz Two heat flow passing through an error signal output On-board 5V supply can be used or shared with MCU 5V	
Specifications	
Model	IBT-2
Input Voltage	6V - 27V
Maximum Current	43A
Input Level	3.3V - 5V
Control Method	PWM or level
Duty Cycle	0% - 100%
Size	4 x 5 x 1.2 cm / 1.6 x 2.0 x 0.5 inch

Table 4.18: Features and Specifications of the IBT-2 Motor Driver Module

4.2.3.1.3 Motor Feedback and Braking Systems

For precise control and safety, the AGV's motors are equipped with encoders and brakes. The two primary motors feature optical incremental encoders and power-off brakes, ensuring accurate feedback and reliable stopping mechanisms.

4.2.3.1.3.1 Encoder Specifications

The motors utilize optical incremental encoders with the following specifications:

Encoder Specifications	
Encoder Type	Optical Incremental
Resolution	1000 PPR (Pulses Per Revolution)
Output Circuit Type	Differential
Output Signal Channels	2

Table 4.19: Encoder Specifications

These encoders provide high-resolution feedback on motor position and speed, enabling precise control and navigation for the AGV.

4.2.3.1.3.2 Brake Specifications

The motors are equipped with power-off brakes, which engage automatically in the event of a power failure, ensuring the AGV remains stationary. The brake specifications are as follows:

Brake Specifications	
Brake Type	`Power Off Brake`
Holding Torque	`200 Ncm`
Brake Rated Voltage	`24V`
Brake Current	`0.16A`
Brake Power	`4W`
Response Time	`50ms`

Table 4.20: Electromagnetic Brake Specifications

The brakes shown in Figure 4.50 provide a holding torque of 200 Ncm, ensuring the AGV remains stable even under load. he fast response time of 50ms guarantees quick engagement, enhancing safety during operation.

Figure 4.50: Electromagnetic Brake for Nema 23

The encoder feedback is integrated into the AGV's control system, enabling real-time monitoring and adjustment of motor performance. The brakes are designed to work seamlessly with the power supply, ensuring immediate activation during emergencies or power loss.

4.2.4 Sensors

Various sensors, such as LiDAR, cameras, and infrared sensors, are utilized for navigation, obstacle detection, and localization.

The AGV is equipped with a variety of sensors to ensure accurate navigation, obstacle detection, weight measurement, and environmental perception. These sensors work together to enhance the robot's autonomy, safety, and efficiency. The following sensors are integrated into the system:

4.2.4.1 VL53L0X (Time-of-Flight Distance Sensor)

The VL53L0X time-of-flight (ToF) distance sensors play an important role in enhancing the AGV's perception of its immediate surrounding in an operational perspective. hese sensors are strategically integrated into the AGV design by deploying eight ToF sensors—two at each corner of the robot insuring a comprehensive coverage. The ToF sensors are especially important in scenarios where traditional sensing methods, such as LiDAR, may be limited or rendered ineffective. For instance, when the AGV carries a lifted platform that obstructs the LiDAR's field of view, the ToF sensors take over to provide reliable distance measurements within a range of 0.5m to 12m .

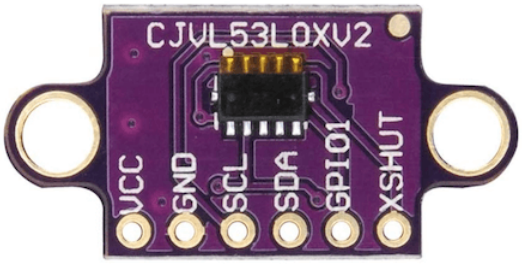


Figure 4.51: VL53L0X time-of-flight (ToF) distance sensor

4.2.4.2 Weight Sensor - Load Sensor 50Kg

This sensor provides accurate and continuous monitoring of the load on the AGV.

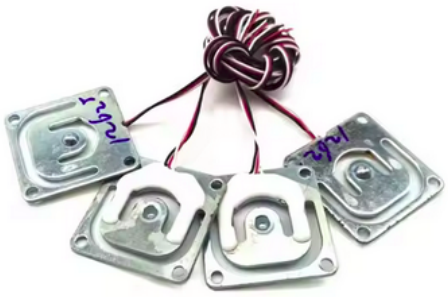


Figure 4.52: Load cell

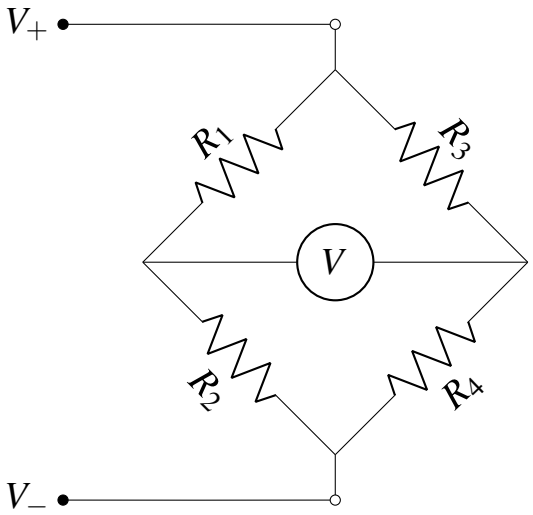


Figure 4.53: Load Cell Circuit with Wheatstone Bridge and Amplifier

The idea behind the load cell is to use the Wheatstone bridge (fig. 4.53). In a Wheatstone bridge, $R_1 = R_2 = R_3 = R_4$, and the output voltage is zero when no force is applied. When weight is applied, it bends the material, which changes the resistivity of the material. This change disrupts the equality of the resistances, causing a difference in voltage across V. By experimenting and calibrating with different weights and tracking the changes in voltage, we can measure the weight.

It is also worth noting that the signal produced depends on the material properties. If the bending factor is linear, the output will be proportional to the applied weight. However, if the bending becomes exponential (which would be a limitation of the material), the relationship between the applied weight and the output signal may no longer be accurate. Additionally, the placement of the load affects the measurement. The voltage difference across V is used to measure the output of the bridge.

In theory, we could use the microcontroller's analog input to read the voltage differences. However, the changes are too small to provide an accurate reading. For this reason, the **HX711** amplifier is used to amplify the signal and convert it into a digital signal.

The HX711 in fig. 4.54 is specifically designed to enhance the weak output signals from the load cellin fig. 4.52, ensuring accurate and reliable weight measurements.

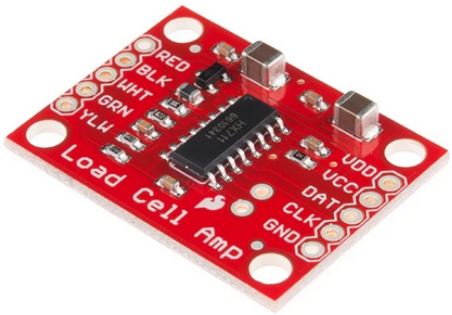


Figure 4.54: HX711 Load cell amplifier

4.2.4.3 RPLIDAR - 360

The RPLIDAR sensor provides comprehensive environmental mapping, making it indispensable for AGV navigation from an electrical control perspective. With its ability to deliver high-accuracy, real-time 360-degree scanning, the RPLIDAR enables the AGV to perceive its surroundings with exceptional precision.

Figure 4.55: RPLIDAR - 360

4.2.4.4 Camera HUSKYLENS

The **HUSKYLENS** vision sensor enhances the AGV's performance used for **object detection and line following**. It provides accurate real-time visual feedback, allowing dynamic adjustments to motor speed, direction, and trajectory.

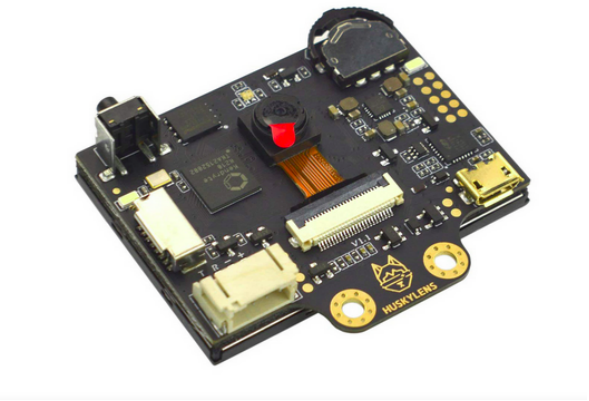


Figure 4.56: Huskylens camera

4.2.4.5 IMU Sensor

The Inertial Measurement Unit (IMU) is used for achieving precise positioning and movement control in automated guided vehicles (AGVs) The MPU9255 integrates a 3-axis gyroscope, 3-axis accelerometer, and 3-axis magnetometer, providing comprehensive data on angular velocity, linear acceleration, and magnetic field orientation. The MPU9255 communicates with the AGV's control system via I2C or SPI interfaces, ensuring low-latency data transmission while minimizing power consumption—a key consideration for energy-efficient operation.

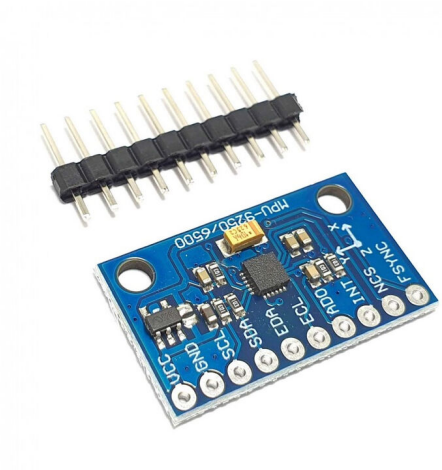


Figure 4.57: Inertia measurement unit 9 axis

4.2.4.6 Barcode Scanner GM65

Figure 4.58: Qr code scanner

4.2.4.7 QTRX-HD-11RC

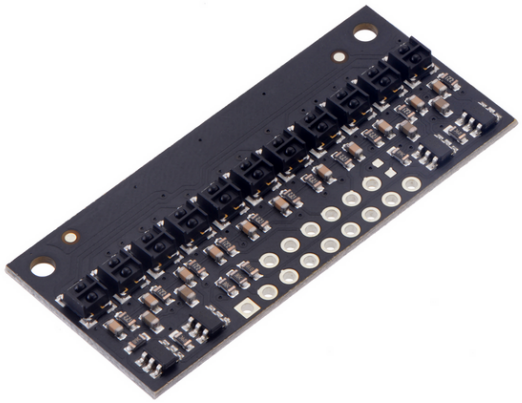


Figure 4.59: QTRX-HD-11RC Reflectance Sensor Array: 11-Channel

4.2.5 Microcontroller and Communication

When selecting a microcontroller (MCU), several key factors must be considered to ensure it meets the demands of the system. One of the most critical criteria is processing power. The system design requires an embedded computer capable of running Linux, which is essential for implementing the Robot Operating System (ROS).

ROS provides a robust framework for handling robotic applications, including sensor fusion, navigation, and communication between various components. Running ROS efficiently requires an embedded system with sufficient computational capabilities, including a high-performance CPU and adequate RAM.

Another crucial factor is the number of available I/O pins. The system involves interfacing with multiple sensors and actuators, necessitating a microcontroller with a high pin count to accommodate all necessary connections. Many of these devices operate using different communication protocols, such as I2C, SPI, and UART.

4.2.5.1 Microcontroller and Embedded computer integration

The AGV's microcontroller and communication system form the backbone of its control and data exchange capabilities, ensuring efficient processing, seamless communication, and real-time control of the robot's operations.

In this design, the **Raspberry Pi 4** shown in Figure fig. 4.60 serves as the primary computational unit, acting as the "brain" of the system. Equipped with a Linux operating system, it hosts all necessary **ROS (Robot Operating System)** packages, enabling high-level decision-making, sensor data processing, and trajectory planning.

4.2.5.1.1 Raspberry Pi 4

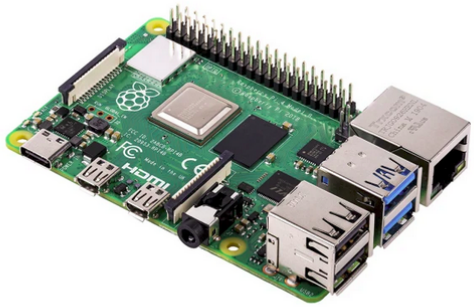


Figure 4.60: Raspberry Pi 4

Complementing the Raspberry Pi, the **ESP32 microcontroller** in fig. 4.61 supports it by managing low-level tasks such as motor control and real-time sensor interfacing. The ESP32 directly interfaces with motor drivers to regulate speed, and direction, ensuring precise actuation of the AGV's propulsion system.

Communication between the Raspberry Pi 4 and the ESP32 is achieved through a **serial communication protocol**, which ensures reliable and efficient data exchange. This division of responsibilities – high-level computation on the Raspberry Pi and low-level control on the ESP32 – optimizes the system's performance and reliability.

Additionally, the sensors are shared between both components, with the ESP32 handling time-sensitive data acquisition and the Raspberry Pi performing higher-level processing and integration into the ROS framework.

4.2.5.1.2 ESP32-S3-DevKitC-1

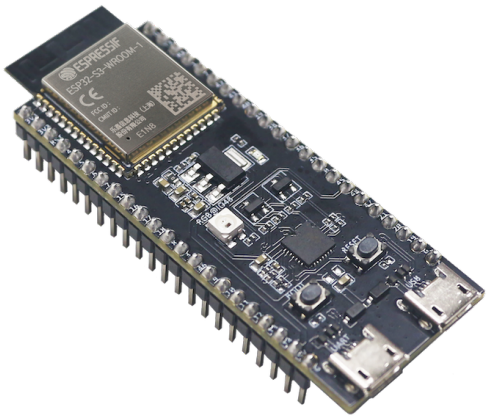


Figure 4.61: ESP32-S3-DevKitC-1

4.2.5.1.3 RS232 to Bluetooth Series Adapter

RS232 to Bluetooth Series Adapter

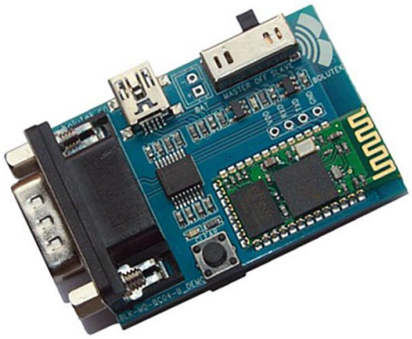


Figure 4.62: RS232 to Bluetooth Series Adapter

4.2.5.1.4 TP-Link TL-WR840N

- The **TP-Link TL-WR840N** is utilized as an access point to facilitate communication between the control unit (PC) and the robot's microcontrollers (Raspberry Pi and ESP32 modules).

- By configuring the TL-WR840N as an access point, it creates a wireless network that enables seamless data and command exchange between the PC, Raspberry Pi, and ESP32 modules.
- This setup allows for remote control and monitoring of the AGV's operations, enhancing its usability and adaptability.

All **sensors, microcontrollers, and motor drivers** in the AGV are selected to work in a **cohesive and well-integrated system**, ensuring seamless communication and efficient operation. **Protocols such as I2C and UART** enable reliable data exchange between components, allowing real-time monitoring and control. This structured communication enhances **precision, synchronization, and responsiveness**, optimizing the AGV's navigation, lifting, and traction systems while maintaining operational efficiency and safety.

4.3 ROS AND SIMULATION

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, [22] To effectively approach ROS programming, it's essential to first understand the foundational concepts of ROS and its package management system. We will explore key ROS components, including the ROS master, nodes, parameter server, messages, and services. Along the way, we will also discuss the necessary steps for installing ROS and initiating work with the ROS master.
In the subsequent sections, we will explore the following topics:

- Why ROS?
- the ROS filesystem level.
- ROS computation graph level.
- ROS community level.

Technical requirements

To follow this chapter, the only thing you need is a standard computer running Ubuntu 20.04 LTS or a Debian 10 GNU/Linux distribution.

Figure 4.63: Ubuntu 20.04

4.3.1 Why ROS?

[1] *Robot Operating System (ROS)* is a flexible framework that provides various tools and libraries for writing robotic software. It offers several powerful features to help developers in tasks such as message passing, distributed computing, code reusing, and implementing state-of-the-art algorithms for robotic applications. The ROS project was started in 2007 by Morgan Quigley and its development continued at Willow Garage, a robotics research lab for developing hardware and open source software for robots. The goal of ROS was to establish a standard way to program robots while offering off-the-shelf software components that can be easily integrated with custom robotic applications. There are many reasons to choose ROS as a programming framework, and some of them are as follows:

4.3.1.1 High-end capabilities:

ROS comes with ready-to-use functionalities. For example, the *Simultaneous Localization and Mapping (SLAM)* and *Adaptive Monte Carlo Localization (AMCL)* packages in ROS can be used for having autonomous navigation in mobile robots, while the MoveIt package can be used for motion planning for robot manipulators. These capabilities can directly be used in our robot software without any hassle. In several cases, these packages are enough for having core robotics tasks on different platforms. Also, these capabilities are highly configurable; we can fine-tune each one using various parameters.

4.3.1.2 Tons of tools:

The ROS ecosystem is packed with tons of tools for debugging, visualizing, and having a simulation. The tools, such as *rqt_gui, RViz*, and *Gazebo*, are some of the strongest open source tools for debugging, visualization, and simulation. A software framework that has this many tools is very rare.

4.3.1.3 Support for high-end sensors and actuators:

ROS allows us to use different device drivers and the interface packages of various sensors and actuators in robotics. Such high-end sensors include 3D LIDAR, laser scanners, depth sensors, actuators, and more. We can interface these components with ROS without any hassle.

4.3.1.4 Inter-platform operability:

The ROS message-passing middleware allows communication between different programs. In ROS, this middleware is known as nodes. These nodes can be programmed in any

language that has ROS client libraries. We can write high-haveance nodes in C++ or C and other nodes in Python or Java.

4.3.1.5 Modularity:

One of the issues that can occur in most standalone robotic applications is that if any of the threads of the main code crash, the entire robot application can stop. In ROS, the situation is different; we are writing different nodes for each process, and if one node crashes, the system can still work.

4.3.1.6 Concurrent resource handling:

Handling a hardware resource via more than two processes is always a headache. Imagine that we want to process an image from a camera for face detection and motion detection; we can either write the code as a single entity that can do both, or we can write a single-threaded piece of code for concurrency. If we want to add more than two features to threads, the application behavior will become complex and difficult to debug. But in ROS, we can access devices using ROS topics from the ROS drivers. Any number of ROS nodes can subscribe to the image message from the ROS camera driver, and each node can have different functionalities. This can reduce the complexity in computation and also increase the debugging ability of the entire system.

Most high-end robotics companies are now porting their software to ROS. This trend is also visible in industrial robotics, in which companies are switching from proprietary robotic applications to ROS. Now that we know why it is convenient to study ROS, we can start introducing its core concepts. There are mainly three levels in ROS: the filesystem level, the computation graph level, and the community level. We will briefly have a look at each level.

4.3.2 The ROS filesystem level

ROS is more than a development framework. We can refer to ROS as a meta-OS, since it offers not only tools and libraries but even OS-like functions, such as hardware abstraction, package management, and a developer toolchain. Like a real operating system, ROS files are organized on the hard disk in a particular manner, as depicted in the following diagram: Here are the explanations for each block in the filesystem:

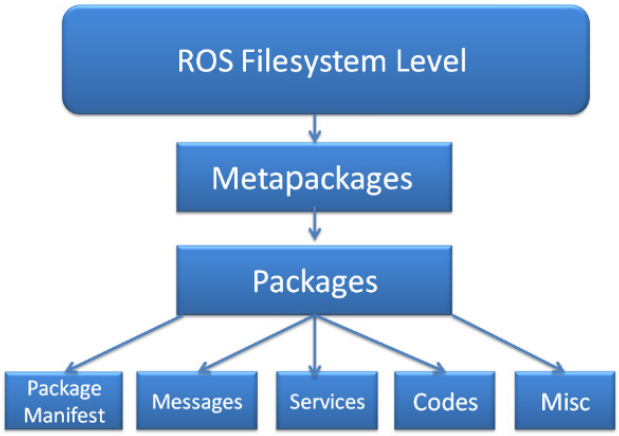


Figure 4.64: ROS filesystem level

- *Packages:* The ROS packages are a central element of the ROS software. They contain one or more ROS programs (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build and release items in the ROS software.

- *Package manifest:* The package manifest file is inside a package and contains information about the package, author, license, dependencies, compilation flags, and so on. The package.xml file inside the ROS package is the manifest file of that package.

- *Metapackages:* The term metapackage refers to one or more related packages that can be loosely grouped. In principle, metapackages are virtual packages that don't contain any source code or typical files usually found in packages.

- *Metapackages manifest:* The metapackage manifest is similar to the package manifest, with the difference being that it might include packages inside it as runtime dependencies and declare an export tag.

- *Messages (.msg):* We can define a custom message inside the msg folder inside a package (my_package/msg/MyMessageType.msg). The extension of the message file is .msg.

- *Services (.srv):* The reply and request data types can be defined inside the srv folder inside the package (my_package/srv/MyServiceType.srv).
- *Repositories:* Most of the ROS packages are maintained using a Version Control System (VCS) such as Git, Subversion (SVN), or Mercurial (hg). A set of files placed on a VCS represents a repository.

The following screenshot gives you an idea of the files and folders of a package that we are going to create in the upcoming sections:

```
ros_pkg
├── action
│   └── demo.action
├── CMakeLists.txt
├── include
│   └── ros_pkg
│       └── demo.h
├── msg
│   └── message.msg
├── src
│   └── demo.cpp
└── srv
    └── service.srv
```

Figure 4.65: List of files inside the package

4.3.3 ROS packages

The typical structure of a ROS package is shown here:

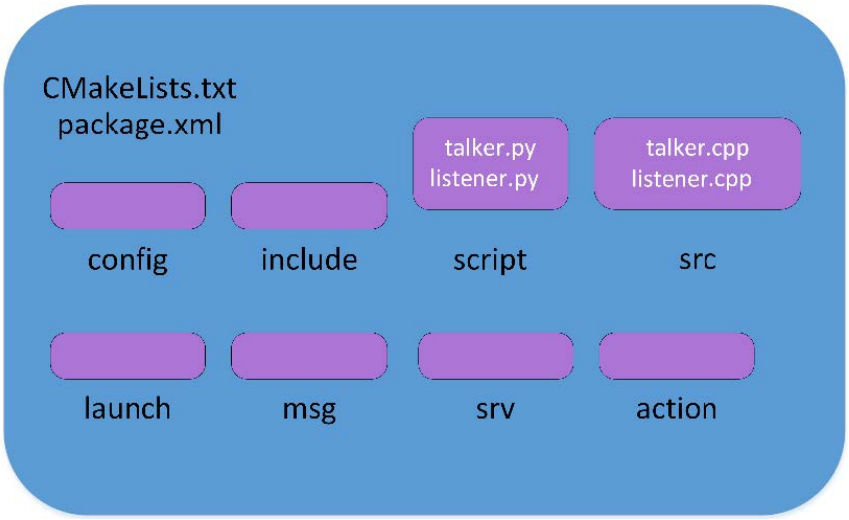


Figure 4.66: Structure of a typical C++ ROS package

- *config:* All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and it is a common practice to name the folder config as this is where we keep the configuration files.

- *include/package_name:* This folder consists of headers and libraries that we need to use inside the package. script: This folder contains executable Python scripts. In the block diagram, we can see two example scripts.
- *src:* This folder stores the C++ source codes.
- *launch:* This folder contains the launch files that are used to launch one or more ROS nodes.
- *msg:* This folder contains custom message definitions.
- *srv:* This folder contains the services definitions.
- *action:* This folder contains the action files.
- *package.xml:* This is the package manifest file of this package.
- *CMakeLists.txt:* This file contains the directives to compile the package.

We need to know some commands for creating, modifying, and working with ROS packages. Here are some of the commands we can use to work with ROS packages:

- *catkin_create_pkg:* This command is used to create a new package.
- *rospack:* This command is used to get information about the package in the filesystem.
- *catkin_make:* This command is used to build the packages in the workspace.
- *rosdep:* This command will install the system dependencies required for this package.

To work with packages, ROS provides a bash-like command called rosbash (http://wiki.ros.org/rosbash), which can be used to navigate and manipulate the ROS package. Here are some of the rosbash commands:

- *roscd:* This command is used to change the current directory using a package name, stack name, or a special location. If we give the argument a package name, it will switch to that package folder.
- *roscp:* This command is used to copy a file from a package.
- *rosed:* This command is used to edit a file using the vim editor.
- *rosrun:* This command is used to run an executable inside a package.

The definition of package.xml in a typical package is shown in the following code:

Code Block 4.3.1: Typical ROS Package.xml

```
<?xml version="1.0"?>
<package>
<name>hello world</name>
<vrsion>0.0.1</version>
<description>The hello world package</description>
<maintainer email="example@gmail.com">example</maintainer>
<buildtool_depend>catkin</buildtool_depend>
<buildtool_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
<export>
</export> </package>
```

The package.xml file also contains information about the compilation.
The <build_ depend></build_depend> tag includes the packages that are necessary for building the source code of the package. The packages inside the <run_depend></run_ depend> tags are necessary for running the package node at runtime.

4.3.4 ROS metapackages

Metapackages are specialized packages that require only one file; that is, a package.xml file. Metapackages simply group a set of multiple packages as a single logical package. In the package.xml file, the metapackage contains an export tag, as shown here:

Code Block 4.3.2: Defining a ROS Metapackage with the package.xml File

```
<export>
<metapackage/>
</export>
```

Also, in metapackages, there are no <buildtool_depend> dependencies for catkin; there are only <run_depend> dependencies, which are the packages that are grouped inside the metapackage.

The ROS navigation stack is a good example of somewhere that contains metapackages. If ROS and its navigation package are installed, we can try using the following command by switching to the navigation metapackage folder:

Code Block 4.3.3: Navigating to the ROS Navigation Metapackage with roscd

```
roscd navigation
```

Open package.xml using your text editor (gedit, in the following case):

Code Block 4.3.4: Opening the package.xml File Using a Text Editor

```
gedit package.xml
```

This is a lengthy file; here is a stripped-down version of it:

Code Block 4.3.5: Structure of the package.xml metapackage

```
<?xml version="1.0"?>
<package>
<name>navigation</name>
<version>1. 14. 0</version>
<description>
A 2D navigation stack that takes in information from
 ↪  odometry, sensor
streams, and a goal pose and outputs safe velocity
 ↪  commands that are sent
to a mobile base.
</description>
<url>http : //wiki.ros.org/navigation</url>
<buildtool_depend>catkin</buildtool_depend>
<run_depend>amcl</run_depend>
...
<export>
<metapackage/>
</export>
</package>
```

This file contains several pieces of information about the package, such as a brief description, its dependencies, and the package version.

4.3.5 ROS messages

ROS nodes can write or read data of various types. These different types of data are described using a simplified message description language, also called ROS messages. These data type descriptions can be used to generate source code for the appropriate message type in different target languages. Even though the ROS framework provides a large set of robotic-

specific messages that have already been implemented, developers can define their own message type inside their nodes. The message definition can consist of two types: `fields` and `constants`. The field is split into field types and field names. The field type is the data type of the transmitting message, while the field name is the name of it.

Here is an example of message definitions:

Code Block 4.3.6: Example of ROS Message Definitions: Fields and Data Types

```
int32 number

string name

float32 speed
```

Here, the first part is the field type and the second is the field name. The field type is the data type, and the field name can be used to access the value from the message. For example, we can use msg.number to access the value of the number from the message.

Here is a table showing some of the built-in field types that we can use in our message:

Built-in Field Types for Message Definition

Primitive type	Serialization	C++	Python
bool (1)	Unsigned 8-bit int	uint8_t (2)	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int (3)
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string (4)	std::string	string
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Table 4.21: Primitive types and their serialization in C++ and Python.

ROS provides a set of complex and more structured message files that are designed to cover a specific application's necessity, such as exchanging common geometrical (geometry_msgs) or sensor (sensor_msgs) information. These messages are composed of different primitive types. A special type of ROS message is called a message header. This header can carry information, such as time, frame of reference or frame_id, and sequence number. Using the header, we will get numbered messages and more clarity about which component is sending the current message. The header information is mainly used to send data such as robot joint transforms. Here is the definition of the header:

Code Block 4.3.7: ROS Message Header: Sequence, Timestamp, and Frame ID

```
uint32 seq

time stamp

string frame_id
```

The rosmsg command tool can be used to inspect the message header and the field types. The following command helps view the message header of a particular message:

```
rosmsg show std_msgs/Header
```

This will give you an output like the preceding example's message header. We will look at the `rosmsg` command and how to work with custom message definitions later in this chapter(section 4.3.9).

4.3.6 The ROS services

ROS services are a type of request/response communication between ROS nodes. One node will send a request and wait until it gets a response from the other. Similar to the message definitions when using the .msg file, we must define the service definition in another file called .`srv`, which must be kept inside the `srv` subdirectory of the package.

An example service description format is as follows:

Code Block 4.3.8: ROS Services: Request and Response

```
#Request message type
string req
---
#Response message type
string res
```

The first section is the message type of the request, which is separated by —, while the next section contains the message type of the response. In these examples, both Request and Response are strings.

4.3.7 The ROS computation graph level

Computation in ROS is done using a network of ROS nodes. This computation network is called the computation graph. The main concepts in the computation graph are ROS **nodes**, **master**, **parameter server**, **messages**, **topics**, **services**, and **bags**. Each concept in the graph is contributed to this graph in different ways.

The ROS communication-related packages, including core client libraries, such as roscpp and rospython, and the implementation of concepts, such as topics, nodes, parameters, and services, are included in a stack called ros_comm (http://wiki.ros.org/ros_comm).
This stack also consists of tools such as rostopic, rosparam, rosservice, and rosnode to introspect the preceding concepts.
The ros_comm stack contains the ROS communication middleware packages, and these packages are collectively called the **ROS graph layer**.

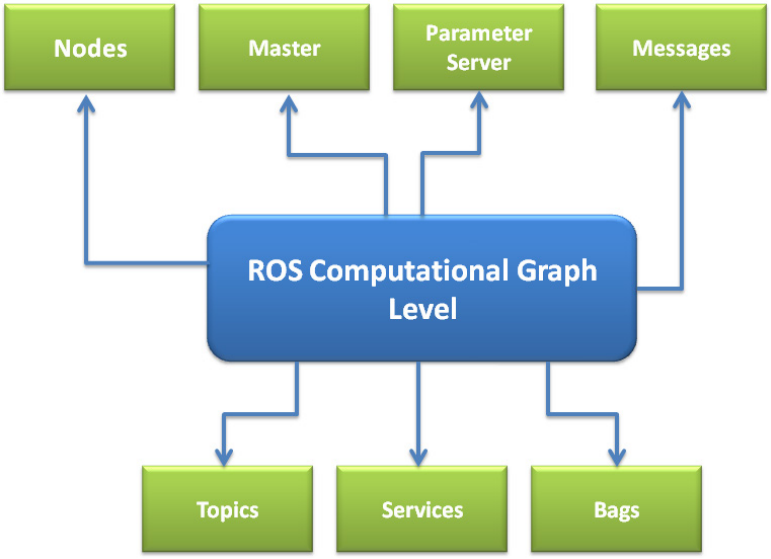


Figure 4.67: Structure of the ROS graph layer

4.3.7.1 Nodes

Nodes are the processes that have computation. Each ROS node is written using ROS client libraries. Using client library APIs, we can implement different ROS functionalities, such as the communication methods between nodes, which is particularly useful when the different nodes of our robot must exchange information between them. One of the aims of ROS nodes is to build simple processes rather than a large process with all the desired functionalities. Being simple structures, ROS nodes are easy to debug.

4.3.7.2 Master

The ROS master provides the name registration and lookup processes for the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS master. In a distributed system, we should run the master on one computer; then, the other remote nodes can find each other by communicating with this master.

4.3.7.3 Parameter server

The parameter server allows you to store data in a central location. All the nodes can access and modify these values. The parameter server is part of the ROS master.

4.3.7.4 Topics

Each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the node is publishing a topic. When a node receives a message through a topic, then we can say that the node is subscribing to a

topic. The publishing node and subscribing node are not aware of each other's existence. We can even subscribe to a topic that might not have any publisher. In short, the production of information and its consumption are decoupled. Each topic has a unique name, and any node can access this topic and send data through it so long as they have the right message type.

4.3.7.5 Logging

ROS provides a logging system for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms. These are known as bagfiles. Bagfiles are very useful features when we're working with complex robot mechanisms.

The following graph shows how the nodes communicate with each other using topics:

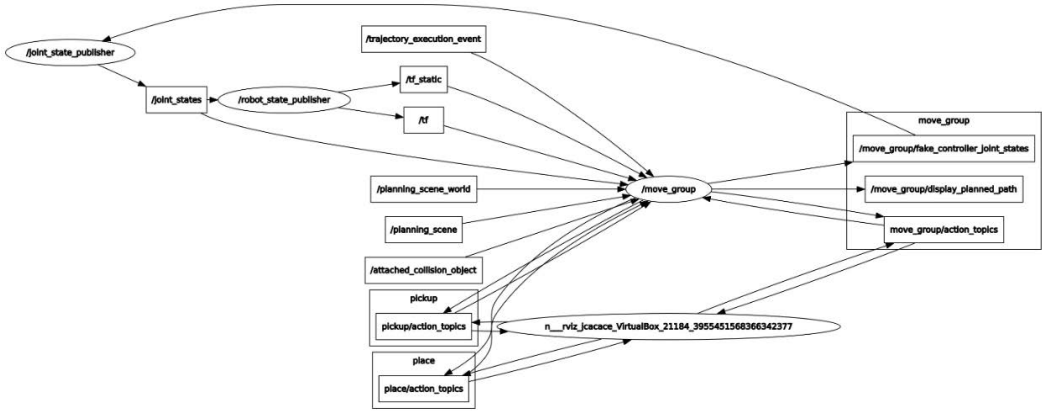


Figure 4.68: Graph of communication between nodes using topics

The topics are represented by rectangles, while the nodes are represented by ellipses. The messages and parameters are not included in this graph. These kinds of graphs can be generated using a tool called **rqt_graph** (http://wiki.ros.org/rqt_graph).

4.3.8 ROS nodes

ROS nodes have computations using ROS client libraries such as `roscpp` and `rospy`. A robot might contain many nodes; for example, one node processes camera images, one node handles serial data from the robot, one node can be used to compute odometry, and so on.

Using nodes can make the system fault-tolerant. Even if a node crashes, an entire robot system can still work. Nodes also reduce complexity and increase debug-ability compared to monolithic code because each node is handling only a single function.

All running nodes should have a name assigned to help us identify them. For example, /camera_node could be the name of a node that is broadcasting camera images.

There is a rosbash tool for introspecting ROS nodes. The `rosnode` command can be used to gather information about an ROS node. Here are the usages of **rosnode**:

- `rosnode info [node_name]`: This will print out information about the node.
- `rosnode kill [node_name]`: This will kill a running node.
- `rosnode list`: This will list the running nodes.
- `rosnode machine [machine_name]`: This will list the nodes that are running on a particular machine or a list of machines.
- `rosnode ping`: This will check the connectivity of a node.
- `rosnode cleanup`: This will purge the registration of unreachable nodes.

some example nodes that use the roscpp client and discuss how ROS nodes that use functionalities such ROS topics, service, messages, and actionlib work.

4.3.9 ROS messages

messages are simple data structures that contain field types. ROS messages support standard primitive data types and arrays of primitive types.

We can access a message definition using the following method. For example, to access std_msgs/msg/String.msg when we are using the roscpp client, we must include std_msgs/String.h for the string message definition.

In addition to the message data type, ROS uses an MD5 checksum comparison to confirm whether the publisher and subscriber exchange the same message data types.

ROS has a built-in tool called `rosmmsg` for gathering information about ROS messages. Here are some parameters that are used along with `rosmmsg`:

- `rosmmsg show [message_type]`: This shows the message's description.
- `rosmmsg list`: This lists all messages.
- `rosmmsg md5 [message_type]`: This displays md5sum of a message.
- `rosmmsg package [package_name]`: This lists messages in a package.
- `rosmmsg packages [package_1] [package_2]`: This lists all packages that contain messages.

4.3.10 ROS topics

Using topics, the ROS communication is unidirectional. Differently, if we want a direct request/response communication, we need to implement ROS services. The ROS nodes communicate with topics using a TCP/IP-based transport known as **TCPROS**. This method is the default transport method used in ROS. Another type of communication is **UDPROS**, which has low latency and loose transport and is only suited for teleoperations. The ROS topic tool can be used to gather information about ROS topics. Here is the syntax of this command:

- **rostopic bw /topic**: This command will display the bandwidth being used by the given topic.
- **rostopic echo /topic**: This command will print the content of the given topic in a human-readable format. Users can use the -p option to print data in CSV format.
- **rostopic find /message_type**: This command will find topics using the given message type.
- **rostopic hz /topic**: This command will display the publishing rate of the given topic.
- **rostopic info /topic**: This command will print information about an active topic.
- **rostopic list**: This command will list all the active topics in the ROS system.
- **rostopic pub /topic message_type args**: This command can be used to publish a value to a topic with a message type.
- **rostopic type /topic**: This will display the message type of the given topic.

4.3.11 ROS services

In ROS services, one node acts as a ROS server in which the service client can request the service from the server. If the server completes the service routine, it will send the results to the service client. For example, consider a node that can provide the sum of two numbers that has been received as input while implementing this functionality through an ROS service. The other nodes of our system might request the sum of two numbers via this service. In this situation, topics are used to stream continuous data flows.

The ROS service definition can be accessed by the following method. For example, `my_package/srv/Image.srv` can be accessed by `my_package/Image`.

In ROS services, there is an MD5 checksum that checks in the nodes. If the sum is equal, then only the server responds to the client.

There are two ROS tools for gathering information about the ROS service. The first tool is **rossrv**, which is similar to **rosmmsg**, and is used to get information about service types. The next command is **rosservice**, which is used to list and query the running ROS services.

Let's explain how to use the rosservice tool to gather information about the running services:

- **rosservice call /service args**: This tool will call the service using the given arguments.
- **rosservice find service_type**: This command will find the services of the given service type.
- **rosservice info /services**: This will print information about the given service.
- **rosservice list**: This command will list the active services running on the system.
- **rosservice type /service**: This command will print the service type of a given service.
- **rosservice uri /service**: This tool will print the service's ROSRPC URI.

4.3.12 ROS bagfiles

The **rosbag** command is used to work with rosbag files. A bag file in ROS is used for storing ROS message data that's streamed by topics. The .bag extension is used to represent a bag file.

Bag files are created using the **rosbag record** command, which will subscribe to one or more topics and store the message's data in a file as it's received. This file can play the same topics that they are recorded from, and it can remap the existing topics too.

Here are the commands for recording and playing back a bag file:

- **rosbag record [topic_1] [topic_2] -o [bag_name]**: This command will record the given topics into the bag file provided in the command. We can also record all topics using the -a argument.
- **rosbag play [bag_name]**: This will play back the existing bag file.

The full, detailed list of commands can be found by using the following command in a Terminal:

Code Block 4.3.9: ROS Bag Files: Recording and Playing Back Topic Data

```
        rosbag play -h
```

There is a GUI tool that we can use to handle how bag files are recorded and played back called `rqt_bag`. To learn more about `rqt_bag`, go to `https://wiki.ros.org/rqt_bag`.

4.3.13 The ROS master

The ROS master is much like a DNS server, in that it associates unique names and IDs to the ROS elements that are active in our system. When any node starts in the ROS system, it will start looking for the ROS master and register the name of the node in it. So, the ROS master has the details of all the nodes currently running on the ROS system. When any of the node's details change, it will generate a callback and update the node with the latest details. These node details are useful for connecting each node.

When a node starts publishing to a topic, the node will give the details of the topic, such as its name and data type, to the ROS master. The ROS master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, the ROS master will share the node details of the publisher to the subscriber node. After getting the node details, these two nodes will be connected. After connecting to the two nodes, the ROS master has no role in controlling them. We might be able to stop either the publisher node or the subscriber node according to our requirements. If we stop any nodes, they will check in with the ROS master once again. This same method is used for the ROS services.

As we've already stated, the nodes are written using ROS client libraries, such as `roscpp` and `rospy`. These clients interact with the ROS master using **XML Remote Procedure Call (XMLRPC)**-based APIs, which act as the backend of the ROS system APIs.

The `ROS_MASTER_URI` environment variable contains the IP and port of the ROS master. Using this variable, ROS nodes can locate the ROS master. If this variable is wrong, communication between the nodes will not take place. When we use ROS in a single system, we can use the IP of a localhost or the name `localhost` itself. But in a distributed network, in which computation is done on different physical computers, we should define `ROS_MASTER_URI` properly; only then will the remote nodes be able to find each other and communicate with each other. We only need one master in a distributed system, and it should run on a computer in which all the other computers can ping it properly to ensure that remote ROS nodes can access the master.

The following diagram(fig. 4.69) shows how the ROS master interacts with publishing and subscribing nodes, with the publisher node publishing a string type topic with a `Hello World` message and the subscriber node subscribing to this topic: When the publisher node starts advertising the `Hello World` message in a particular topic, the ROS master gets the details of the topic and the node. It will check whether any node is subscribing to the same topic. If no nodes are subscribing to the same topic at that time, both nodes will remain unconnected. If the publisher and subscriber nodes run at the same time, the ROS master will exchange the details of the publisher to the subscriber, and they will connect and exchange data through ROS topics.

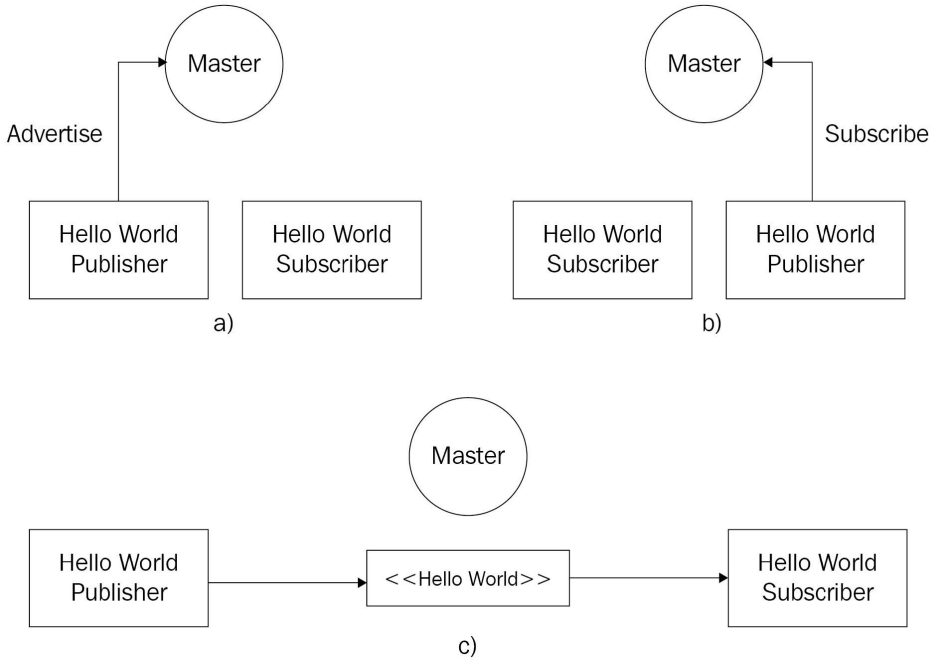


Figure 4.69: Communication between the ROS master and Hello World publisher and subscriber

4.3.14 ROS parameter

When programming a robot, we might have to define robot parameters to tune our control algorithm, such as the robot controller gains P, I, and D of a standard proportional integral derivative controller. When the number of parameters increases, we might need to store them as files. In some situations, these parameters must be shared between two or more programs. In this case, ROS provides a parameter server, which is a shared server in which all the ROS nodes can access parameters from this server. A node can read, write, modify, and delete parameter values from the parameter server.

We can store these parameters in a file and load them into the server. The server can store a wide variety of data types and even dictionaries. The programmer can also set the scope of the parameter; that is, whether it can be accessed by only this node or all the nodes.

The parameter server supports the following XMLRPC data types:

- 32-bit integers
- Booleans
- Strings
- Doubles
- ISO8601 dates

- Lists
- Base64-encoded binary data

We can also store dictionaries on the parameter server. If the number of parameters is high, we can use a YAML file to save them. Here is an example of the YAML file parameter definitions:

```
/camera/name : 'nikon' #string_type
/camera/fps : 30 #integer
/camera/exposure : 1.2 #float
/camera/active : true #boolean
```

The `rosparam` tool is used to get and set the ROS parameter from the command line. The following are the commands for working with ROS parameters:

- `rosparam set [parameter_name] [value]`: This command will set a value in the given parameter.
- `rosparam get [parameter_name]`: This command will retrieve a value from the given parameter.
- `rosparam load [YAML_file]`: The ROS parameters can be saved into a YAML file. It can load them into the parameter server using this command.
- `rosparam dump [YAML_file]`: This command will dump the existing ROS parameters into a YAML file.
- `rosparam delete [parameter_name]`: This command will delete the given parameter.
- `rosparam list`: This command will list existing parameter names.

These parameters can be changed dynamically when you're executing a node that uses these parameters by using the **dynamic_reconfigure** package (`http://wiki.ros.org/dynamic_reconfigure`).

4.3.15 ROS distributions

ROS updates are released with new ROS distributions. A new distribution of ROS is composed of an updated version of its core software and a set of new/updated ROS packages. ROS follows the same release cycle as the Ubuntu Linux distribution: a new version of ROS is released every 6 months. Typically, for each Ubuntu LTS version, an **LTS** version of ROS is released. **Long Term Support (LTS)** and means that the released software will be maintained for a long time (5 years in the case of ROS and Ubuntu).

Built-in Field Types for Message Definition

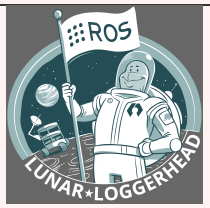
Distro	Release date	Poster	Turtle	EOL date
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			June 27, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)

Table 4.23: ROS Distributions Table

4.3.16 Running the ROS master and the ROS parameter server

Before running any ROS nodes, we should start the ROS master and the ROS parameter server. We can start the ROS master and the ROS parameter server by using a single command called **roscore**, which will start the following programs:

- `ROS master`
- `ROS parameter server`
- `rosout logging nodes`

The rosout node will collect log messages from other ROS nodes and store them in a log file, and will also re-broadcast the collected log message to another topic. The /rosout topic is published by ROS nodes using ROS client libraries such as roscpp and rospy, and this topic is subscribed by the rosout node, which rebroadcasts the message in another topic called /rosout_agg. This topic contains an aggregate stream of log messages. The roscore command should be run as a prerequisite to running any ROS nodes. The following screenshot shows the messages that are printed when we run the roscore command in a Terminal.

Use the following command to run `roscore` on a Linux Terminal:

Code Block 4.3.10: Starting the ROS Master

```
roscore
```

After running this command, we will see the following text in the Linux Terminal:

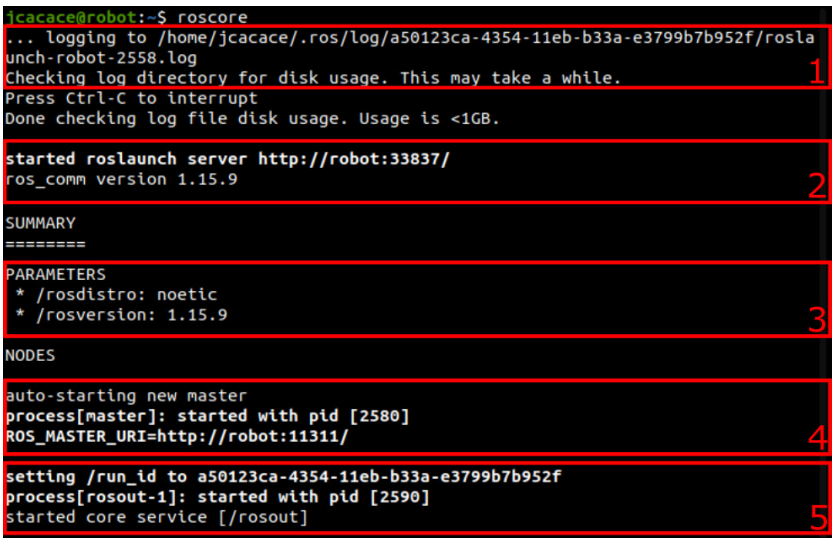


Figure 4.70: Terminal messages while running the roscore command

- In section 1, we can see that a log file is created inside the ~/.ros/log folder for collecting logs from ROS nodes. This file can be used for debugging purposes.
- In section 2, the command starts a ROS launch file called `roscore.xml`. When a launch file starts, it automatically starts `rosmaster` and the ROS parameter server. The `roslaunch` command is a Python script, which can start `rosmaster` and the ROS parameter server whenever it tries to execute a launch file. This section shows the address of the ROS parameter server within the port.
- In section 3, we can see parameters such as `rosdistro` and `rosversion` being displayed in the Terminal. These parameters are displayed when it executes `roscore.xml`. We will look at `roscore.xml` in more detail in the next section.
- In section 4, we can see that the `rosmaster` node is being started with `ROS_MASTER_URI`, which we defined earlier as an environment variable.
- In section 5, we can see that the `rosout` node is being started, which will start subscribing to the `/rosout` topic and rebroadcasting it to `/rosout_agg`.

The following is the content of `roscore.xml`:

Code Block 4.3.11: Output of the roscore Command and the Structure of roscore.xml

```
<launch>
    <group ns="/">
        <param name="rosversion" command="rosversion roslaunch" />
        <param name="rosdistro" command="rosversion -d" />
        <node pkg="rosout" type="rosout" name="rosout"
         ↪  respawn="true"/>
    </group>
</launch>
```

When the `roscore` command is executed, initially, the command checks the command-line argument for a new port number for `rosmaster`. If it gets the port number, it will start listening to the new port number; otherwise, it will use the default port. This port number and the `roscore.xml` launch file will be passed to the `roslaunch` system. The `roslaunch` system is implemented in a Python module; it will parse the port number and launch the `roscore.xml` file.

In the `roscore.xml` file, we can see that the ROS parameters and nodes are encapsulated in a group XML tag with a `/` namespace. The group XML tag indicates that all the nodes inside this tag have the same settings.

The `rosversion` and `rosdistro` parameters store the output of the `rosversion roslaunch` and `rosversion -d` commands using the `command` tag, which is a part of the ROS param tag. The `command` tag will execute the command mentioned in it and store the output of the command in these two parameters.

`rosmaster` and the parameter server are executed inside `roslaunch` modules via the `ROS_MASTER_URI` address. This happens inside the `roslaunch` Python module.
`ROS_MASTER_URI` is a combination of the IP address and port that `rosmaster` is going to listen to. The port number can be changed according to the given port number in the `roscore` command.

4.3.16.1 Checking the roscore command's output

Let's check out the ROS topics and ROS parameters that are created after running roscore. The following command will list the active topics in the Terminal:

Code Block 4.3.12: Listing Active Topics After Running the roscore Command

```
rostopic list
```

The list of topics is as follows, as per our discussion of the rosout node's subscribe /rosout topic. This contains all the log messages from the ROS nodes. /rosout_agg will rebroadcast the log messages:

Code Block 4.3.13: Active Topics After Running roscore: /rosout and /rosout_agg

```
/rosout
/rosout_agg
```

The following command lists the parameters that are available when running roscore. The following command is used to list the active ROS parameter:

Code Block 4.3.14: Listing Active ROS Parameters After Running roscore

```
rosparam list
```

These parameters are mentioned here; they provide the ROS distribution name, version, the address of the roslaunch server, and run_id, where run_id is a unique ID associated with a particular run of roscore:

Code Block 4.3.15: ROS Parameters Available After Running roscore

```
/rosdistro
/roslaunch/uris/host_robot_virtualbox_51189
/rosversion
/run_id
```

The list of ROS services that's generated when running roscore can be checked by using the following command:

Code Block 4.3.16: Listing ROS Services Generated by roscore

```
rosservice list
```

The list of services that are running is as follows:

Code Block 4.3.17: Active ROS Services After Running roscore

```
/rosout/get_loggers
/rosout/set_logger_level
```

These ROS services are generated for each ROS node, and they are used to set the logging levels.

4.3.17 Robot modeling using URDF

The Unified Robot Description Format (URDF) is an XML file format used to describe the physical properties of a robot in robotics applications, in particular within the Robot Operating System (ROS) ecosystem. It specifies the robot structure (links, joints, sensors and actuators etc.)

URDF can represent the kinematic and dynamic description of the robot, the visual representation of the robot, and the collision model of the robot. The following tags are the commonly used URDF tags to compose a URDF robot model:

4.3.17.1 link:

The link tag represents the single link of a robot. Using this tag, we can model a robot link and its properties. The modeling includes the size, the shape, and the color; it can even import a 3D mesh to represent the robot link. We can also provide the dynamic properties of the link, such as the inertial matrix and the collision properties. The syntax is as follows:

Code Block 4.3.18: Defining a Robot Link in URDF

```
<link name="<name of the link>">
    <inertial>...........</inertial>
    <visual> ...........</visual>
    <collision>..........</collision>
</link>
```

The following is a representation of a single link. The Visual section represents the real link of the robot, and the area surrounding the real link is the Collision section. The Collision section encapsulates the real link to detect a collision before hitting the real link:

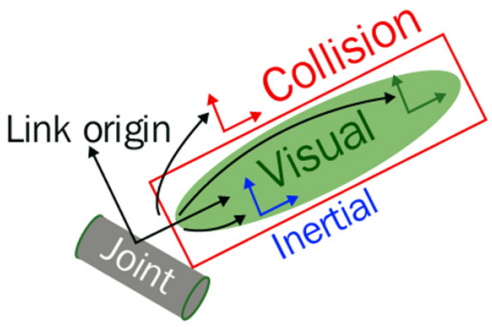


Figure 4.71: A visualization of the URDF link [1]

4.3.17.2 joint:

The *joint* tag represents a robot joint. We can specify the kinematics and dynamics of the joint and set the limits of the joint movement and its velocity. The joint tag supports the different types of joints, such as *revolute, continuous, prismatic, fixed, floating,* and *planar*. The syntax is as follows:

Code Block 4.3.19: Defining a Robot joint in URDF

```
<joint name="<name of the joint>">
<parent link="link1"/>
<child link="link2"/>
<calibration .... />
<dynamics damping = ..../>
<limit effort = ..../>
</joint>
```

A URDF joint is formed between two links; the first is called the Parent link, and the second is called the Child link. Note that a single joint can have a single parent and multiple children at the same time. The following is an illustration of a joint and its links:

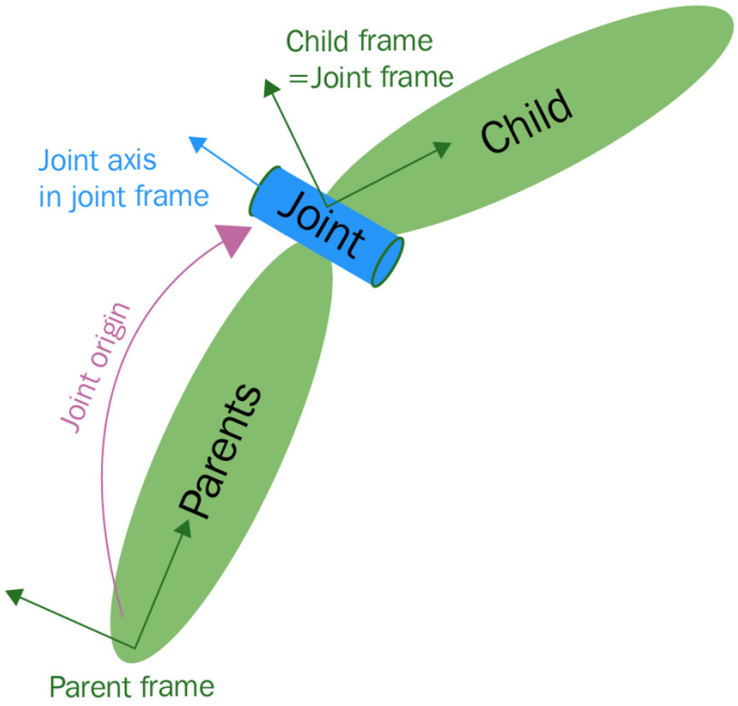


Figure 4.72: A visualization of the URDF joint [1]

4.3.17.3 robot:

This tag encapsulates the entire robot model that can be represented using URDF. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot. The syntax is as follows:

Code Block 4.3.20: Defining a Robot Model in URDF

```
<robot name="<name of the robot>">
<link> ..... </link>
<joint> ..... </joint>
<joint> ........</joint>
</robot>
```

A robot model consists of connected links and joints. Here is a visualization of the robot model:

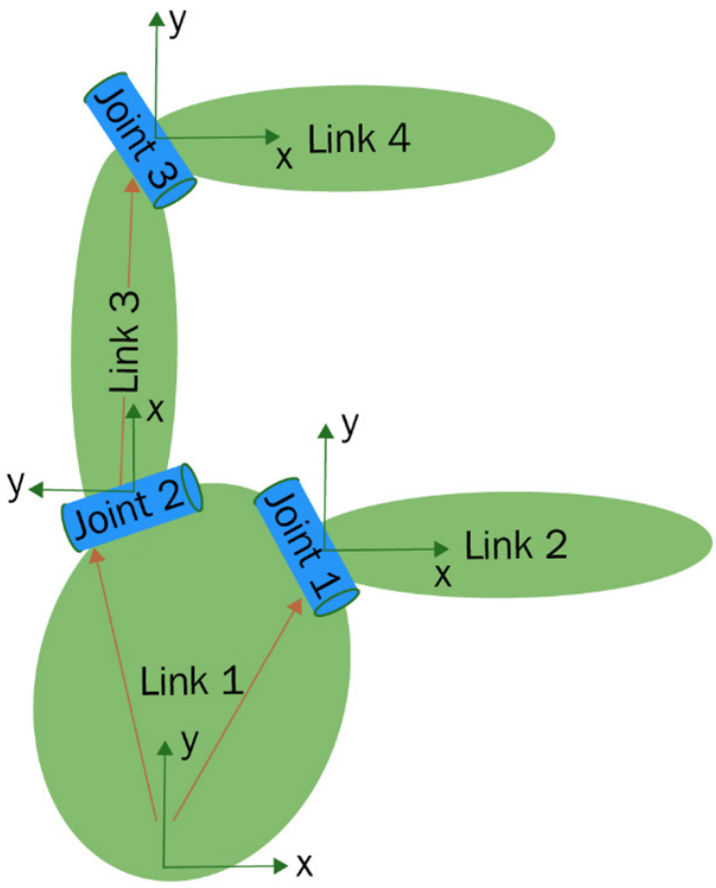


Figure 4.73: A visualization of a robot model with joints and links [1]

4.3.17.4 gazebo:

This tag is used when we include the simulation parameters of the Gazebo simulator inside the URDF. We can use this tag to include gazebo plugins, gazebo material properties, and more. The following shows an example that uses gazebo tags:

Code Block 4.3.21: Defining Gazebo Material Properties in URDF

```
<gazebo reference="link_1">
<material>Gazebo/Black</material>
</gazebo>
```

You can find more about URDF tags at wiki.ros [23]. We are now ready to use the elements listed earlier to create a new robot from scratch. In the next section, we are going to create a new ROS package containing a description of the different robots.

4.3.17.5 Adding physical and collision properties to a URDF model

Before simulating a robot in a robot simulator, such as Gazebo or CoppeliaSim, we need to define the robot link's physical properties, such as geometry, color, mass, and inertia, as well as the collision properties of the link. Good robot simulations can be obtained only if the robot dynamic parameters (for instance, its mass, inertia, and more) are correctly specified in the urdf file. In the following code, we include these parameters as part of the base_link:

Code Block 4.3.22: Defining Physical and Collision Properties for a Robot Link in URDF

```
<link name="base_link">
...
<collision>
  <geometry>
      <box size="0.9 0.85 0.5"/>
  </geometry>
    <origin xyz="0.0 0.0 0.025" rpy="0.0 0.0 0.0"/>
</collision>
<inertial>
<origin xyz="0 0 0" rpy="0 0 0"/>
<mass value="35"/>
<inertia ixx="3.2" ixy="0.0" ixz="0.0" iyy="4" iyz="0.0"
 ↪  izz="$3.9"/>
</inertial>  </link>
```

4.3.17.6 Transmission:

transmission tag is used to model the relationship between a robot's joints and actuators (such as motors, servos, or other mechanical devices that provide motion). It defines how power or motion is transferred from an actuator to a joint in the robot's structure. The ¡transmission¿ tag usually works with ¡joint¿ and ¡actuator¿ tags to specify how an actuator controls the motion of the robot's joints:

Code Block 4.3.23: Modeling Joint-Actuator in URDF with "transmission"

```
<robot name="example_robot">

<!-- Define the joint -->
<joint name="joint_1" type="revolute">
  <parent link="base_link"/>
  <child link="link_1"/>
  <axis xyz="0 0 1"/>
  <limit effort="10.0" velocity="1.0" lower="-1.57" upper="1.57"/>
</joint>
<!-- Define the transmission -->
<transmission name="trans_1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint_1"/>
  <actuator name="motor_1">
                <!-- Gear ratio -->
    <mechanicalReduction>100.0</mechanicalReduction>
  </actuator>
</transmission>
<!-- Define the actuator (motor) -->
<gazebo>
  <plugin name="motor_1_plugin"
   ↪  filename="libgazebo_motor_plugin.so">
    <motor_type>electric</motor_type>
    <max_power>100.0</max_power>
  </plugin>
</gazebo>
</robot>
```

<joint>: The joint joint_1 connects the base_link to link_1 and allows rotational movement. It has limits on effort and velocity. <transmission>: The transmission trans_1 links joint_1 to the actuator (motor motor_1). It uses the SimpleTransmission type.
<mechanicalReduction>: The actuator has a gear reduction of 100:1, meaning for every

100 rotations of the motor, the joint will rotate once. <actuator>: This specifies the motor (motor_1) that will control joint_1. <gazebo>: This block includes a Gazebo plugin (libgazebo_motor_plugin.so) to simulate the motor's behavior in a Gazebo simulation environment, with specific motor properties like motor_type and max_power.

In summary:

Dependencies for URDF Simulation in Gazebo

Step	Component	Effect if Missing
1	Links (`<link>`)	The robot has no physical structure, making it impossible to define shapes, visualize, or interact with physics.
2	Joints (`<joint>`)	The robot will be completely rigid. No movement or articulation between parts is possible.
3	Inertia (`<inertial>`)	Gazebo will generate warnings or unstable behavior because the physics engine relies on mass and inertia properties to simulate motion correctly.
4	Collision (`<collision>`)	The robot will pass through objects and not interact with the environment physically. It may also affect contact-based sensors.
5	Gazebo Plugin (`<plugin>`)	The robot cannot interact with Gazebo's physics, controllers, or sensors. Features like camera feeds, joint control, and custom physics will not work.
6	Gazebo Simulation	The robot cannot be tested in a realistic environment with physics, gravity, and other external forces. It remains a static model without dynamic behavior.
7	Transmission (`<transmission>`)	Motors will not work. Even if controllers are added, they will not be able to apply forces to move the joints.

Table 4.24: Essential Dependencies for Simulating a URDF in Gazebo

4.3.18 Creating URDF model

After learning about URDF and its important tags, we can start some basic modeling using URDF. The robot's URDF model comprises eight rigid links and seven joints, designed to balance geometric simplicity with functional accuracy. The central chassis (base_link), modeled as a rectangular prism, forms the structural foundation. Symmetrically attached to this base are two cylindrical wheels (left_wheel, right_wheel), each connected via continuous rotation joints to enable differential steering. A vertical lifting mechanism, represented as a cylinder (lifting_mechanism), extends upward from the chassis through a prismatic joint, providing linear motion for height adjustment. Rigidly mounted atop this actuator is a box-shaped platform (platform), secured by a fixed joint to ensure stability. Three sensor units—two cameras (camera1, camera2) and a LiDAR (lidar)—are modeled as compact boxes and affixed to the platform through additional fixed joints, ensuring precise perceptual alignment. By prioritizing minimalistic shapes (cylinders for rotational elements, boxes for planar surfaces) and logical joint configurations (two continuous, one prismatic, and four fixed), the design achieves computational efficiency while retaining fidelity to the robot's core mechanical and sensing capabilities.

4.3.18.1 robot modeling using xacro

The flexibility of URDF reduces when we work with complex robot models. Some of the main features that URDF is missing include simplicity, reusability, modularity, and programmability. If someone wants to reuse a URDF block 10 times in their robot description, they can copy and paste the block 10 times. If there is an option to use this code block and make multiple copies with different settings, it will be very useful while creating the robot description. The URDF is a single file and we can't include other URDF files inside it. This reduces the modular nature of the code. All code should be in a single file, which reduces the code's simplicity. Also, if there is some programmability, such as adding variables, constants, mathematical expressions, and conditional statements in the description language, it will be more userfriendly. Robot modeling using xacro meets all of these conditions. Some of the main features of xacro are as follows:

- *Simplify URDF:* xacro is a cleaned-up version of URDF. It creates macros inside the robot description and reuses the macros. This can reduce the code length. Also, it can include macros from other files and make the code simpler, more readable, and more modular.

- *Programmability:* The xacro language supports a simple programming statement in its description. There are variables, constants, mathematical expressions, conditional statements, and more that make the description more intelligent and efficient.

Instead of .urdf, we need to use the .xacro extension for xacro files. Here is an explanation of the xacro code:

Code Block 4.3.24: XML Syntax with Xacro Extensions

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="my_robot">
```

These lines specify a namespace that is needed in all xacro files to parse the xacro file. After specifying the namespace, we need to add the name of the xacro file. In the next section.

4.3.18.2 Using properties

Using xacro, we can declare constants or properties that are the named values inside the xacro file, which can be used anywhere in the code. The main purpose of these constant definitions is that instead of giving hardcoded values on links and joints, we can keep constants, and it will be easier to change these values rather than finding the hardcoded values and then replacing them. An example of using properties is given here. We declare the length and radius of the base link and the pan link. So, it will be easy to change the dimension here rather than changing the values in each one:

Code Block 4.3.25: Xacro Properties to Define Reusable Values for Robot Dimensions

```
<!-- body value -->
<xacro:property name="base_length" value="0.9" />
<xacro:property name="base_width" value="0.85" />
<xacro:property name="base_hight" value="0.5" />
<!-- Wheel values -->
<xacro:property name="wheel_r" value="0.15" />
<xacro:property name="wheel_length" value="0.05" />
<!-- Lidar values  -->
<xacro:property name="lidar_r" value="0.1" />
<xacro:property name="lidar_l" value="0.05" />
```

We can use the value of the variable by replacing the hardcoded value with the following definition(code block 4.3.26):

Code Block 4.3.26: Defining the base link visually

```
<link name="base_link">
  <visual>
```

```
    <geometry>
      <box size="${base_length} ${base_width} ${base_hight}"/>
    </geometry>
      <origin xyz="0.0 0.0 ${base_hight/2.0}" rpy="0.0 0.0 0.0"/>
      <material name="green"/>
  </visual>
```

Here(code block 4.3.26), the value 0.9, is replaced with ”base_length”, and ”0.85” is replaced with
”base_width”.

4.3.18.3 math expression in xacro

We can build mathematical expressions inside $ using basic operations such as +, -, *, /, unary minus, and parentheses. Exponentiation and modulus are not supported yet. The following is a simple math expression used inside the code:

Code Block 4.3.27: Math Expressions in Xacro

```
<link name="platform_link">
<visual>
  <geometry>
    <box size=
    ↪  "${base_length/1.5} ${base_width/1.5} ${base_hight/8}"/>
  </geometry>
    <origin xyz="0.0 0.0 0" rpy="0.0 0.0 0.0"/>
    <material name="green"/>
</visual>
```

4.3.18.4 xacro macros

One of the main features of xacro is that it supports macros. We can use xacro to reduce the length of complex definitions. Here is a xacro definition we used in our code to specify inertial values,

Code Block 4.3.28: Xacro Macros

```
<xacro:macro name="box_inertia" params="m l w h xyz rpy">
<inertial>
   <origin xyz="${xyz}" rpy="${rpy}"/>
   <mass value="${m}"/>
```

```
        <inertia ixx="${ (m/12)*(h*h + l*l) }" ixy="0.0" ixz="0.0"
        ↪  iyy="${(m/12)*(w*w + l*l)}" iyz="0.0"
        ↪  izz="${(m/12) * (w*w + h*h)}"/>
    </inertial>
  </xacro:macro>
```

Here(code block 4.3.28), the macro is named box_inertia, and its parameters are {m ,l ,w ,h ,xyz ,rpy}. we can pass these parameters as a vaues or as a xacro property(code block 4.3.29):

Code Block 4.3.29: Defining box inertia with Xacro

```
<xacro:box_inertia m="5.0" l="${2*base_length}" w="${2*base_width}"
↪  h="${2*base_hight}" xyz="0.0 0.0 ${base_hight/2.0}"
↪  rpy="0.0 0.0 0.0"/>
```

4.3.18.5 Including other xacro files

We can extend the capabilities of the robot xacro by including the xacro definition of sensors using the xacro:include tag. The following code snippet shows how to include a sensor definition in the robot xacro:

Code Block 4.3.30: Including External Xacro Files to Extend Robot Model Definitions

```
<xacro:include filename="$(find ros_robot_pkg)/urdf/definition.xacro"/>
```

Here, we include a xacro file to call the definitions and the constants.

4.3.19 Visualizing the 3D robot model in RViz

After designing the URDF, we can view it on RViz. We can create a launch file and put the following code into the launch folder:

Code Block 4.3.31: Launch File for Visualizing a 3D Robot Model in RViz

```
<launch>
<param name="robot_description" command=
 ↪  "$(find xacro)/xacro $(find the_pkg_name)/.../my_robot.urdf.xacro"
 ↪  />
<node name="robot_state_publisher" pkg="robot_state_publisher"
 ↪  type="robot_state_publisher" />
<node name="rviz" pkg="rviz" type="rviz"
 ↪  args="-d $(find the_pkg_name)/rviz/config.rviz" />
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher"/>
<node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui"
 ↪  type="joint_state_publisher_gui"/>
<!--  Controller Manager  -->
  <node name="controller_spawner" pkg="controller_manager"
   ↪  type="spawner" respawn="false" output="screen"
        args="diff_drive_controller" />
</launch>
```

in code block 4.3.31 The `<launch>` tag is the root tag that defines the entire launch file. All the nodes and parameters that need to be launched are specified within this tag.

The `<param>` tag sets the `robot_description` parameter, which specifies the URDF or XACRO file that contains the robot's description. In this case, the `command` attribute runs the `xacro` tool to process the `my_robot.urdf.xacro` file located in the package specified by `the_pkg_name`.

The `<node>` tag launches the `robot_state_publisher` node. This node reads the robot's description and publishes the state of the robot (e.g., joint positions, transformations) to ROS topics. It uses the `robot_description` parameter that was set earlier.

The next `<node>` tag launches RViz, the visualization tool used to display the robot in a 3D environment. The `args` attribute specifies a pre-configured RViz setup file (`config.rviz`) located in the package `the_pkg_name`. This setup is used for visualizing the robot model and its movements.

The subsequent `<node>` tags launch two joint state publisher nodes.
The `joint_state_publisher` node publishes the joint states (such as the positions of robot joints) to ROS topics. The `joint_state_publisher_gui` node includes a graphical user

interface (GUI) that allows users to interactively control the joint states of the robot.

Finally, the `<node>` tag for the `controller_spawner` node is responsible for spawning and managing robot controllers. The node is part of the `controller_manager` package, and the `spawner` executable is used to spawn a controller, in this case, `diff_drive_controller`. The `respawn` attribute is set to `false`, so the node will not automatically respawn if it crashes. The `output` attribute ensures that the output from the node is printed to the screen.

We can launch the model using the following command(code block 4.3.32):

Code Block 4.3.32: Launching ROS package

```
roslaunch the_pkg_name launch_file_name.launch
```

If everything works correctly, we will get the robot in RViz, as shown here(fig. 4.74):

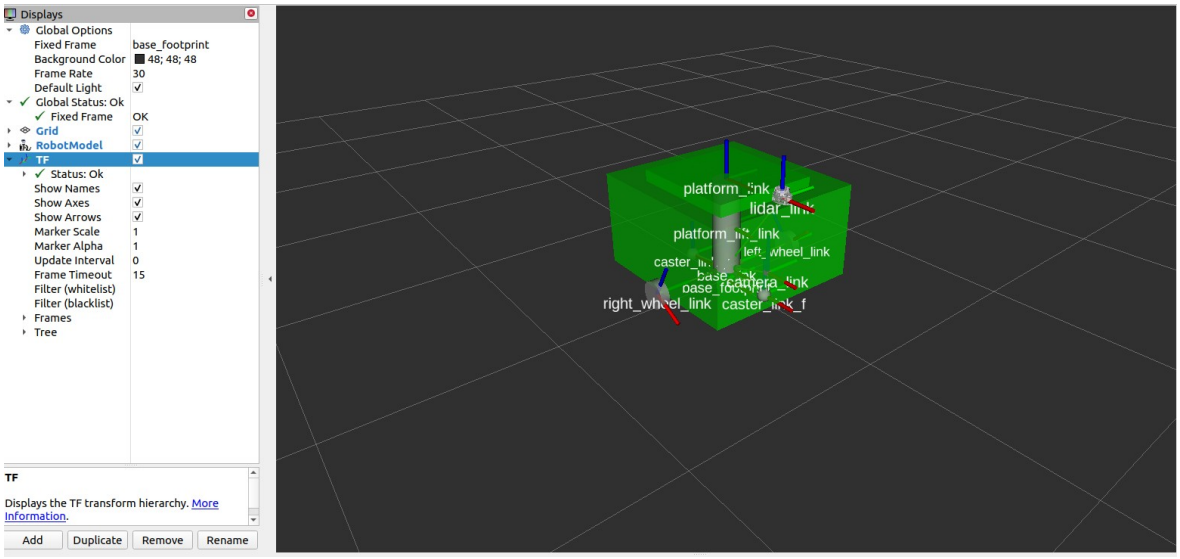


Figure 4.74: A visualization of a robot model with Rviz

4.3.19.1 Interacting with joints in Rviz

In the previous version of ROS, the GUI of `joint_state_publisher` was enabled thanks to a ROS parameter called `use_gui`. To start the GUI in the launch file, this parameter had to be set to true before starting the `joint_state_publisher` node. In the current version of ROS, launch files should be updated to launch `joint_state_ publisher_gui` instead of using `joint_state_publisher` with the `use_gui` parameter.

We can see that an extra GUI came along with RViz; it contains sliders to control the Whell joints and the lifting platform joint. This GUI is called the Joint State Publisher Gui node and belongs to the `joint_state_publisher_gui` package:

Code Block 4.3.33: Joint State Publisher GUI in RViz

```
<node name="joint_state_publisher_gui" pkg="joint_state_
publisher_gui" type="joint_state_publisher_gui" />
```

We can include this node in the launch file using the following statement. The limits of pan-and-tilt should be mentioned inside the joint tag:

Code Block 4.3.34: Prismatic Joint with Motion Limits in URDF for RViz Interaction

```
<joint name="platform_lift_joint" type="prismatic">
    <origin xyz="0.0 0.0 ${base_hight / 2}" rpy="0.0 0.0 0.0"/>
    <parent link="base_link"/>
    <child link="platform_lift_link"/>
    <axis xyz="0 0 1"/> <!-- Movement along the Z-axis -->
    <!-- Motion range and limits -->
    <limit lower="0.0" upper="${base_hight/2}" effort="50"
     ↪  velocity="1.0"/>
</joint>
```

In code block 4.3.34 the ¡limit¿ tag defines the limits of effort, velocity, and angle. In this scenario, effort is the maximum force supported by this joint, expressed in Newton; lower and upper indicate the lower and upper limits of the joint, in radians for the revolute joint and in meters for the prismatic joints. velocity is the maximum joint velocity expressed in m/s.

The following screenshot shows the user interface that is used to interact with the robot joints:

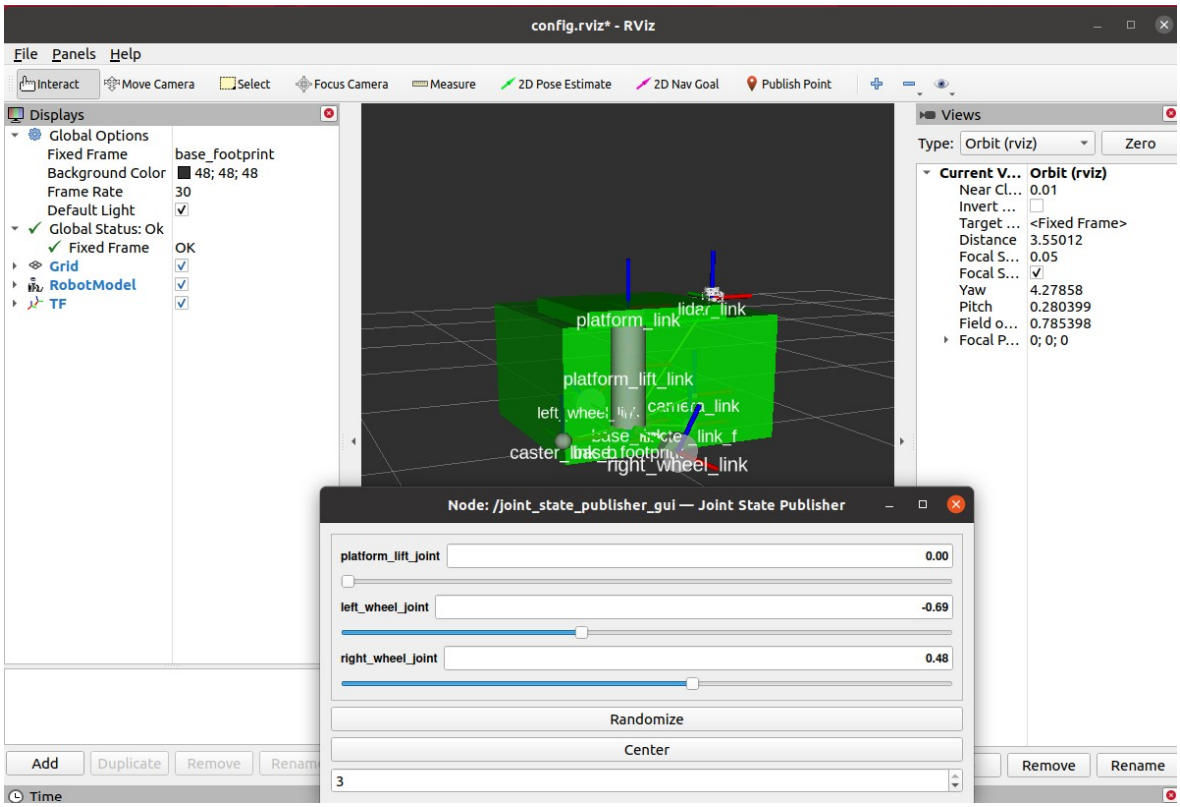


Figure 4.75: The joint level of the platform lifting mechanism

In this user interface, we can use the sliders to set the desired joint values. The basic elements of a urdf file have been discussed. In the next section, we will add additional physical elements to our robot model.

4.3.20 Simulation Environment

4.3.20.1 Introduction to the Simulation Environment

In developing a control strategy for mobile robots such as Automated Guided Vehicles (AGVs), simulation plays a crucial role in testing software components, robot behavior, and control algorithms across various environments. Before physically building the robotic system, running a simulation provides a risk-free and cost-effective approach to refining its design and functionality. By simulating the robot's behavior in a controlled virtual environment, different algorithms can be tested and optimized to ensure efficient navigation, obstacle avoidance, and perception without the risk of hardware damage.

The simulation environment was built using the *Robot Operating System (ROS)* and the *Gazebo* simulator, with the `gazebo_ros` package. `RViz` was utilized for real-time visualization of sensor data and robot trajectories, while `OpenCV` handled computer vision tasks such as *line following* and *QR code detection*. The following sections provide detailed insights into their implementation and role in the project.

4.3.20.2 Tools and Framework

4.3.20.2.1 Gazebo

Gazebo was initially developed in 2002 to facilitate the simulation of ground robot applications in both indoor and outdoor environments [24]. Over the years, it has evolved into a mature open-source project widely adopted by the global robotics community for various applications. Gazebo follows a modular architecture, incorporating four key components essential for robot simulation:

- *Physics Engine Support* – Gazebo integrates multiple physics engines to handle collision detection, contact dynamics, and reaction forces between rigid bodies.
- *Sensor Simulation* – It provides a comprehensive library of commonly used robotic sensors, including cameras, LiDAR, sonar, GPS, and IMUs, along with configurable noise models for realistic sensor emulation.
- *Multiple Interfaces* – The simulator supports various interfaces for programmatic interaction, including C++ for developing plugins.
- *Graphical User Interface (GUI)* – Gazebo features an interactive 3D environment that enables users to visualize and manipulate the simulated world in real-time.

Figure 4.76: Gazebo Simulator

Gazebo is a robust simulation tool integrated with the Robot Operating System (ROS), designed to simulate robotic and sensor applications in both indoor and outdoor 3D environments. It follows a Client-Server architecture coupled with a topic-based Publish/Subscribe model for inter-process communication, enabling efficient data exchange between the various components of the system.

For dynamic simulations, Gazebo leverages several high-performance physics engines, including the Open Dynamics Engine (ODE) [25], Bullet [26], Simbody [27], and the Dynamic Animation and Robotics Toolkit (DART) [28]. These engines handle rigid-body physics simulations, providing highly accurate interactions between objects. Additionally, Gazebo employs the Object-Oriented Graphics Rendering Engine (OGRE) [29] for 3D graphics rendering, generating realistic visual environments that enhance the simulation experience.

In this architecture, the Gazebo Client sends control data, such as the coordinates of simulated objects, to the Server, which then manages real-time control of the virtual robot. Gazebo also supports distributed simulation, allowing the Client and Server to run on separate machines, which can be beneficial for scaling and performance.

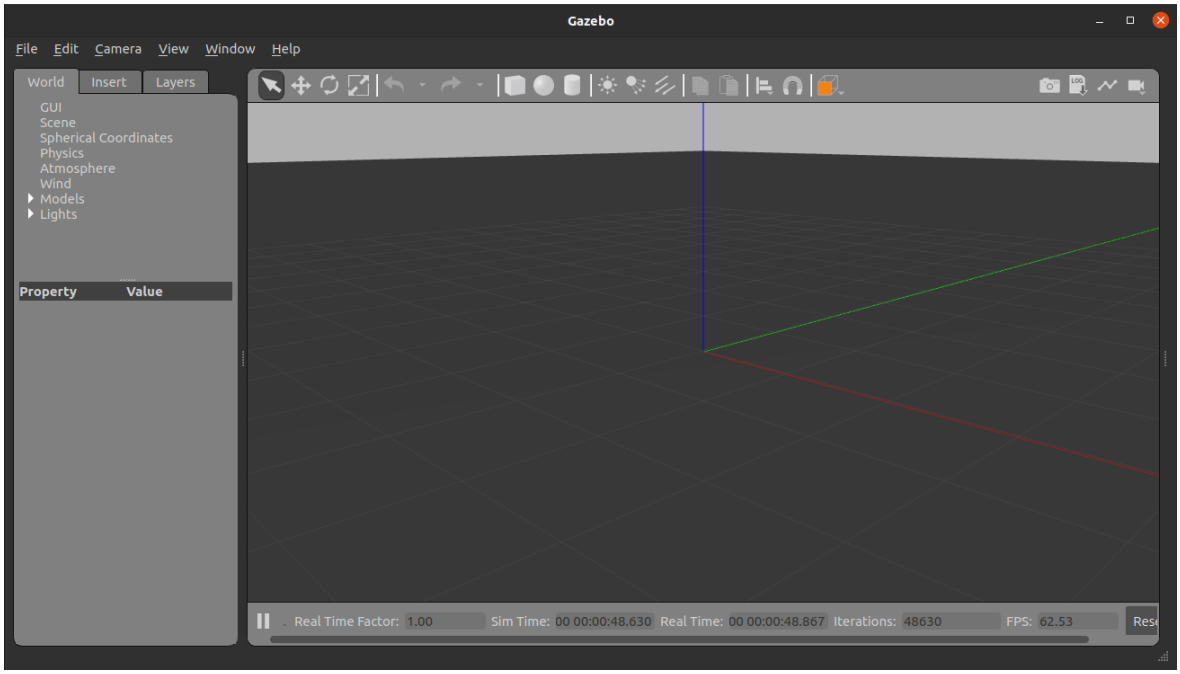


Figure 4.77: Empty world in Gazebo

The ROS Plugin for Gazebo ensures integration between the two platforms, enabling direct communication with ROS. This allows both simulated and real robots to be controlled using the same software interface, making Gazebo an ideal platform for testing, developing, and validating robotic systems in a virtual environment before deployment on physical hardware.

This plugin facilitates bi-directional communication through ROS topics, services, and actions, allowing sensor data from Gazebo—such as LiDAR, cameras, and IMUs—to be published as ROS messages while also enabling ROS-based velocity and trajectory commands to control simulated robots. Additionally, the plugin supports `ros_control`, making it possible to implement position, velocity, and effort controllers, which are essential for tuning PID loops before testing them on real actuators.

Code Block 4.3.35: TurtleBot3 in Gazebo with Customizable Parameters

```
  <launch>
<arg name="model" default="$(env TURTLEBOT3_MODEL)"
 ↪  doc="model type [burger, waffle, waffle_pi]"/>
<arg name="x_pos" default="0.0"/>
<arg name="y_pos" default="0.0"/>
<arg name="z_pos" default="0.0"/>

<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name"
   ↪  value="$(find turtlebot3_gazebo)/worlds/empty.world"/>
  <arg name="paused" value="false"/>
  <arg name="use_sim_time" value="true"/>
  <arg name="gui" value="true"/>
  <arg name="headless" value="false"/>
  <arg name="debug" value="false"/>
</include>

<param name="robot_description" command=
"$(find xacro)/xacro --inorder
$(find turtlebot3_description)/
urdf/turtlebot3_$(arg model).urdf.xacro" />

<node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf"
 ↪  args="-urdf -model turtlebot3_$(arg model) -x $(arg x_pos) -y
$(arg y_pos) -z $(arg z_pos) -param robot_description" />
</launch>
```

The provided example demonstrates how the *ROS Plugin for Gazebo* enables the spawning and control of a robot model within a simulated environment, enabling *smooth and efficient integration* with the *Robot Operating System (ROS)*. The launch file initializes an empty *Gazebo* world and loads a robot model described in a *URDF (Unified Robot Description Format)* file using the `spawn_model` node from the `gazebo_ros` package. Additionally,

the `robot_state_publisher` node is included to publish the robot's joint states, ensuring that its transformations can be visualized and utilized in *ROS-based frameworks* such as *MoveIt* or *RViz*. This integration allows developers to test *robot behaviors, sensor data integration, and control algorithms* in a virtual environment before deploying them on physical hardware.

It facilitates bi-directional communication through *ROS topics, services, and actions*, enabling sensor data from Gazebo—such as *LiDAR, cameras, and IMUs*—to be published as *ROS messages* while also allowing *ROS-based velocity and trajectory commands* to control simulated robots. Additionally, the plugin supports *ros_control*, enabling the implementation of *position, velocity, and effort controllers*, which are essential for tuning *PID loops* before applying them to real actuators.

This specific example shows a *ROS launch file* that spawns a *TurtleBot3* model shown fig. 4.78 in an empty *Gazebo* simulation environment. This file provides flexibility by allowing users to specify the *TurtleBot3 model type* (`burger`, `waffle`, or `waffle_pi`) and configure the *robot's initial position*. It also sets critical simulation parameters, such as *enabling GUI rendering, synchronizing ROS time with Gazebo simulation time, and dynamically loading the robot's description* based on the selected model type. The launch file ensures a well-structured simulation environment where *robotic software, motion planning, and perception algorithms* can be tested under realistic conditions.

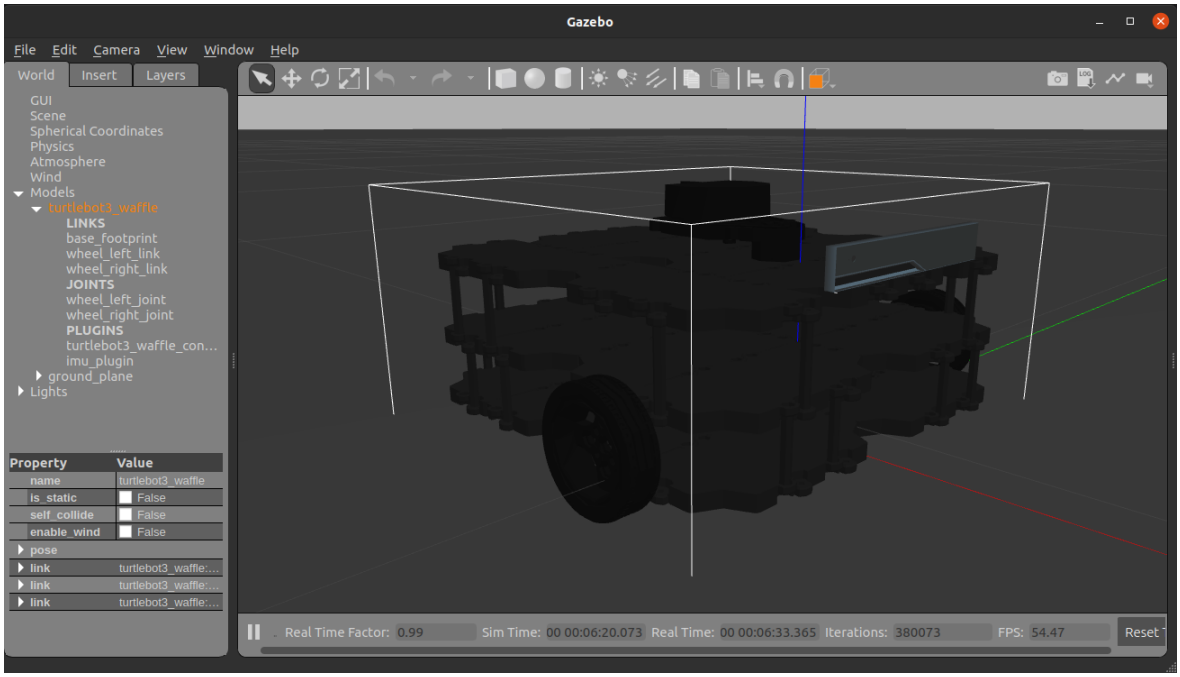


Figure 4.78: TurtleBot3 Waffle spawned in empty world

The Gazebo Graphical User Interface (GUI) provides an interactive environment for users to visualize and manipulate their simulated worlds in real-time. One of the key features

of the GUI is the set of side tabs, which organize various tools and functionalities. Among these, the left side pannel that appears fig. 4.79 World, Insert, and Layers tabs are particularly important for managing the simulation environment and objects within it.

The *World Tab* in Gazebo's GUI allows users to manage and configure the properties of the simulation world, providing several options for modifying the environment settings. Users can adjust the *gravity settings* to control the strength of gravity in the simulation, select the *physics engine* (such as *ODE*, *Bullet*, or *DART*) to determine the level of simulation fidelity and performance, and customize *time and weather conditions*, including factors like sunlight, fog, and rain. Additionally, the World tab provides detailed information about the models within the simulation, including their *joints* and *links*. Users can visualize and modify the connections between different parts of a robot or object, defining how they move and interact with each other. The *links* represent rigid bodies, while *joints* define the relative motion between those bodies, such as revolute, prismatic, or fixed joints. This tab plays a crucial role in fine-tuning the simulation environment and robot behavior, ensuring that the simulation behaves as expected for the application at hand. It is an essential tool for accurately replicating real-world conditions and testing robotic systems in varied scenarios.

The *Insert Tab* allows users to add various objects and components to the simulation environment. Users can insert robot models, lights, sensors, and even other world elements, which helps populate the simulation for testing and development. Additionally, the tab makes it easy to add custom plugins or adjust the robot's attributes in the virtual world.

The *Layers Tab* provides control over the visibility of different elements within the simulation. Users can toggle the visibility of models, sensors, and other objects in the scene to focus on specific parts of the simulation. This helps streamline the workspace, especially when dealing with complex environments and large-scale simulations.

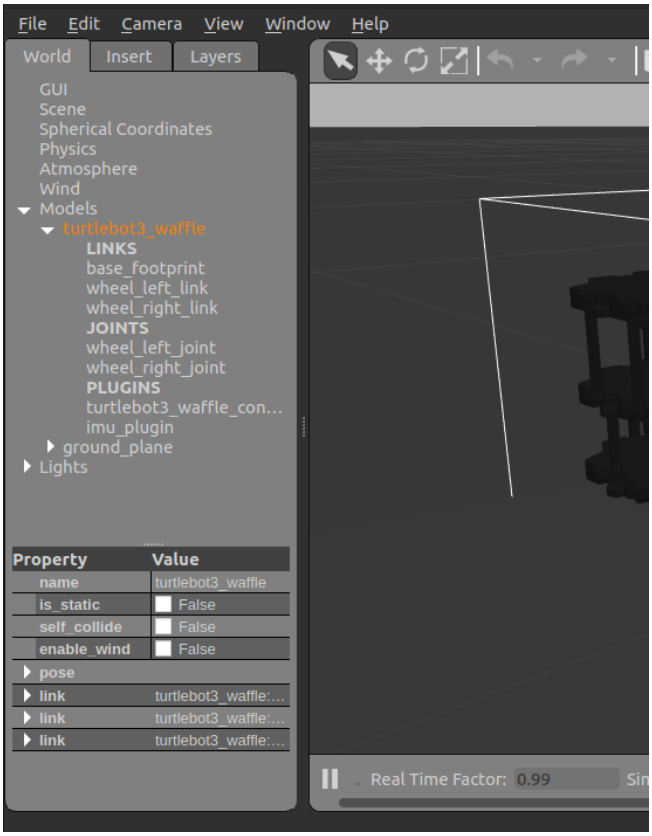


Figure 4.79: Gazebo Graphical User Interface left side pannel

The lower part of the Gazebo GUI displayed in fig. 4.80 provides critical real-time simulation information and performance metrics. One of the most important features is the *Real-Time Factor (RTF)*, which indicates the ratio between real-time and simulated time. This factor helps users determine how efficiently the simulation is running, with values close to 1.0 indicating that the simulation is running in real-time. If the RTF is greater than 1.0, it suggests that the simulation is running faster than real time, while a value less than 1.0 indicates that it is running slower.

In addition to RTF, it also displays *Sim Time*, which shows the simulation's elapsed time. This is useful for tracking the progression of events in the simulation and synchronizing it with other systems. The *Real Iterations* and *Sim Iterations* counters provide insight into the number of iterations performed by the simulation per second for real-time and simulated time, respectively. Monitoring these values helps users assess the efficiency of the simulation and diagnose performance issues if the simulation is not proceeding as expected.

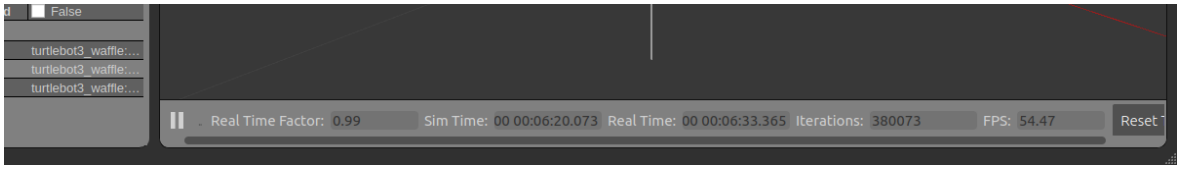


Figure 4.80: Gazebo Graphical User Interface lower part

The top part of the Gazebo GUI (fig. 4.81) provides several essential controls and information related to the simulation's overall operation. At the top-left corner, users will find the *File* menu, which includes options to open, save, and manage Gazebo worlds. It also allows users to load and save configurations, and set preferences for simulation settings.

In addition, the top part of the GUI provides options for manipulating the lighting within the simulation. Users can select from *Point*, *Spot*, and *Directional* lights to illuminate different parts of the world. Point lights emit light in all directions from a single point, Spot lights create a focused beam of light, and Directional lights simulate sunlight, casting parallel rays in a specific direction. This allows for realistic and customizable lighting in the simulation environment.

The *Selection Light* tool enables users to adjust the lighting between two or more objects in the simulation, offering more control over how objects are illuminated and how shadows are cast in the environment.

Moreover, the GUI allows users to switch between different *perspective views*, such as top-down, side, or camera views, to inspect and interact with the simulation from different angles.

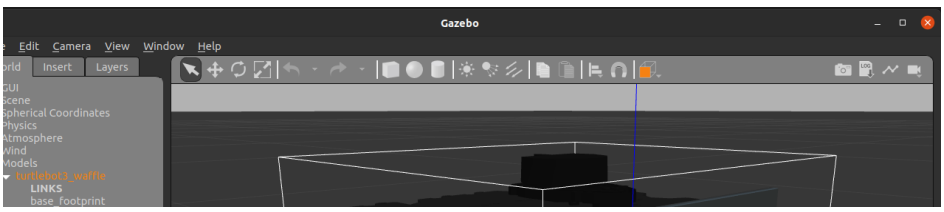


Figure 4.81: Gazebo Graphical User Interface top part

Right-clicking on an object in the Gazebo simulation environment brings up a context-sensitive menu that allows users to perform a variety of actions to interact with and manipulate the object. The options available in this menu include:

1. *Move To*: This option allows users to move a selected object to a specified location in the simulation world. After selecting this option, users can click on a point in the world to which the object will be moved, making it easier to position objects accurately within the environment.

2. *Follow*: The Follow option enables users to make the camera follow a particular object in the simulation. When this option is activated, the camera will automatically track the

movement of the selected object, allowing users to observe its behavior from a fixed perspective, which can be useful when testing robot movements or simulating dynamic scenarios.

3. *Edit*: The Edit option opens a set of interactive tools that allow users to modify the properties of the selected object, such as its position, orientation, size, and material properties.

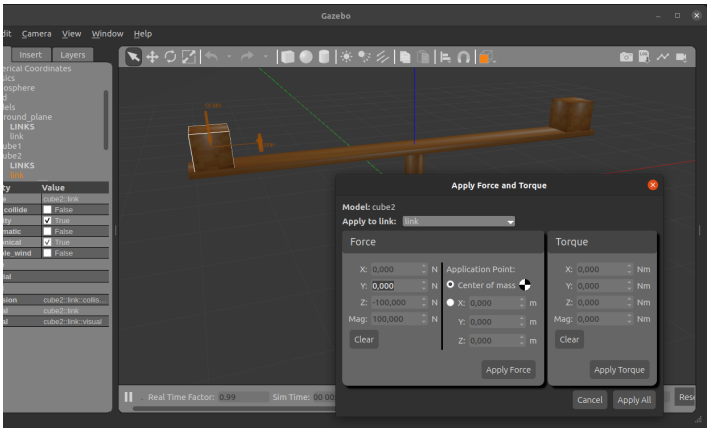


Figure 4.82: Example of force applied on one side of a seesaw in gazebo

4. *Apply Force/Torque*: This option gives users the ability to apply a force or torque to the selected object, which is especially useful for testing how the object reacts under various physical conditions.fig. 4.82 shows how users can specify the direction, magnitude, and duration of the applied force or torque, allowing for controlled experiments on the object's behavior within the simulated environment.

4.3.20.3 RViz

RViz (ROS Visualization) is an open-source 3D visualization tool that provides real-time graphical representations of robotic systems.

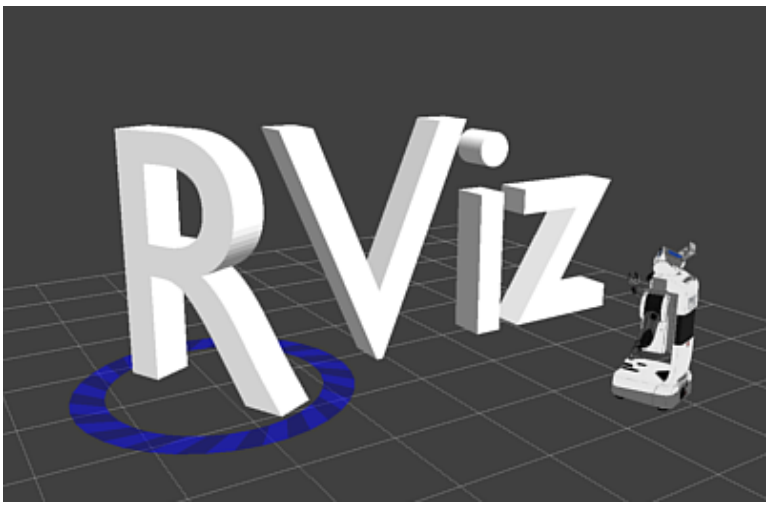


Figure 4.83: Rviz (Ros Visualization)

It serves two primary purposes: (1) visualizing sensor data, such as LiDAR scans, point clouds from 3D sensors (e.g., RealSense, Kinect), and camera images, in an intuitive 3D environment; and (2) enabling interactive control of robotic systems by sending commands like navigation goals or waypoints. In this project, RViz was used extensively to monitor sensor outputs, validate navigation algorithms, and debug the robot's behavior during simulation. Its seamless integration with ROS topics and plugins made it an indispensable tool for visualizing complex data streams, such as laser range finder (LRF) distances and Point Cloud Data (PCD), without requiring additional programming.

4.3.21 Implementation of the Simulation Environment

The implementation of the simulation environment is carefully designed to ensure that the results obtained in simulation closely reflect real-world performance. Most simulation parameters, including the robot's dimensions, mass properties, sensor specifications, and environmental conditions, are selected to match the specifications outlined in the competition framework. This ensures that the robot developed based on simulation results can seamlessly transition from a virtual setting to real-world deployment with minimal modifications.

A key aspect of this simulation is the accurate modeling of robot dynamics, actuator characteristics, and sensor behavior. The physics engine parameters, such as friction coefficients, and sensor noise models, are tuned to replicate real-world conditions as precisely as possible. Additionally, the design of the virtual environment, including terrain features, obstacles, and lighting conditions, is configured to match the competition setup. This allows for realistic testing and validation of navigation and control algorithms.

By incorporating these considerations, the simulation provides a reliable testing ground for developing and refining motion planning, perception, and control strategies. This approach reduces the gap between simulated and real-world performance, ensuring that insights gained from the virtual environment translate effectively to the physical robot. However, like any simulation-based approach, there are inherent advantages and disadvantages when using simulators to test robot behaviors, as discussed in [30].

Despite these limitations, careful selection of simulation parameters—such as physics engine settings, sensor noise models, and environmental conditions—enhances the accuracy of the virtual testing environment. The following subsections detail the implementation of the robot model, world design, and control and sensor integration, outlining how each component contributes to achieving a high-fidelity simulation.

4.3.21.1 World Design

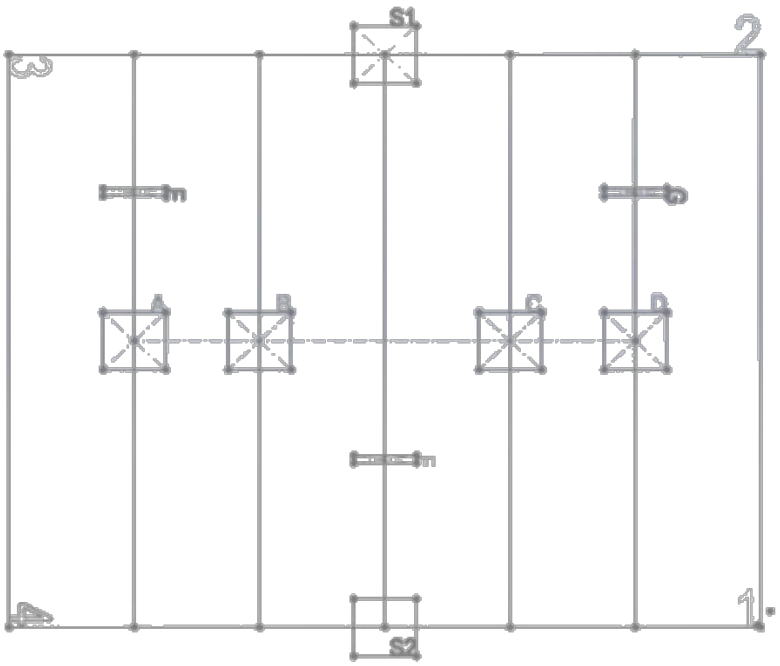


Figure 4.84: Teknofest competition real map layout

The simulation environment was designed to replicate the competition layout in fig. 4.84 as closely as possible. The terrain was constructed using custom models to accurately mimic the competition area, incorporating key features such as a *white wooden floor*, *black lines for path following*, *platforms for load dynamics*, *obstacles*, and *friction variations* to simulate real-world conditions.

In Gazebo, this entire environment is saved as a world file, which includes all the models, textures, and configurations necessary to recreate the simulation. The world file is written in the Simulation Description Format (SDF), an XML-based format used to describe the properties and behavior of objects and environments in Gazebo.

SDF allows for easy definition of physical properties, geometries, lighting, and sensor configurations, making it a versatile and standard way to represent simulation environments. The SDF format ensures that the simulation setup is reproducible, shareable, and can be modified easily for different testing scenarios. Similar to the Unified Robot Description Format (URDF), which is used for defining robot models in ROS, SDF provides a structured way to describe objects and environments. While URDF focuses primarily on robot kinematics and geometry, SDF is more comprehensive and extends its capabilities to encompass the full environment, including physics properties, sensor configurations, and world dynamics.

The world file was made through the design of individual SDF files for each model within

the environment. These individual models were first defined with each having its own dedicated SDF file. Once the models were completed, they were then integrated into a single world file, named `competition_area.world` (in SDF format). This world file serves as the central configuration, bringing together all the models, their properties, and the environment settings.

4.3.21.1.1 Environment Layout

Designed to replicate the competition area closely the world contains custom models were created to represent key elements of the space, ensuring that the robot's interaction with its surroundings is as realistic as possible. Key components of the environment layout include the following:

4.3.21.1.1.1 Flooring and Pathways:

A *white wooden floor* was chosen to resemble the actual competition floor, providing a surface with consistent friction properties for the robot to interact with. The *black lines for path following* were added on the white floor according to the competition specifications, acting as markers for the robot's autonomous navigation algorithms to follow.

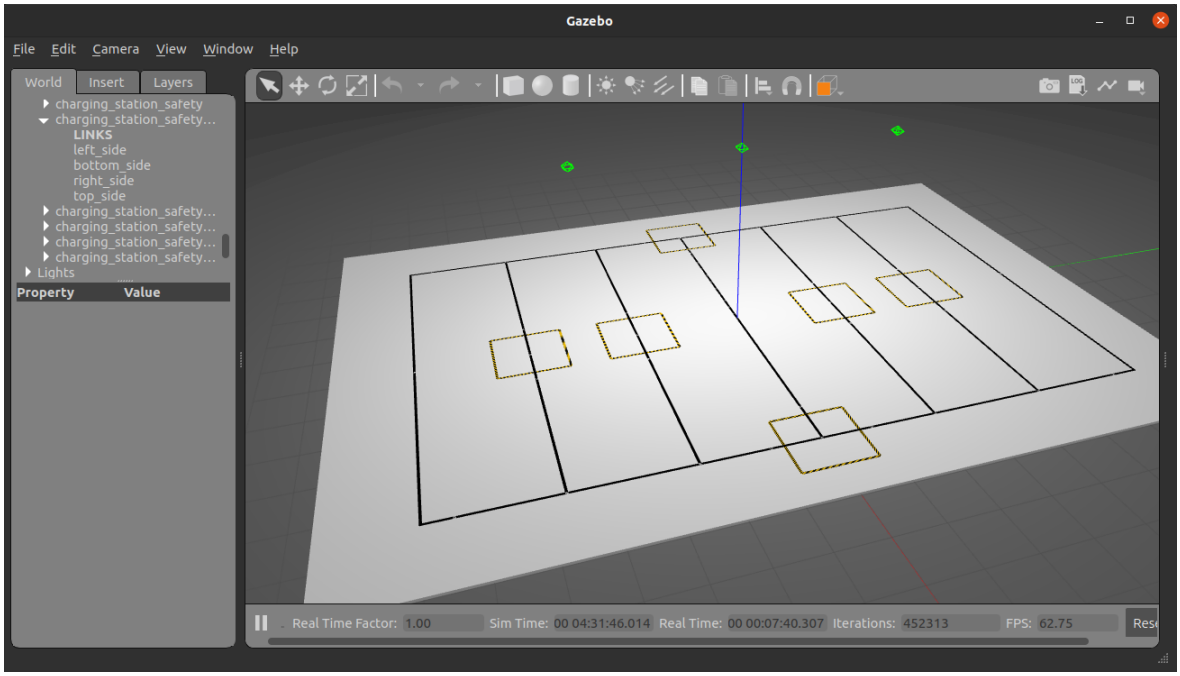


Figure 4.85: White wooden floor with black lines in gazebo

The black lines for path following were also incorporated into the simulation as individual objects within the Gazebo environment. Each line was treated as a separate model to allow easy modifications, such as adjusting their position, length, or orientation, without

affecting the rest of the simulation environment. This modular approach makes it easier to tweak the lines for different test scenarios, ensuring flexibility and quick adjustments during development and testing.

Code Block 4.3.36: Creates multiple black line patterns in the simulation environment.

```
...
<wall_time>1738074256 296636706</wall_time>
<iterations>117145</iterations>

<model name='black_line'>
  <pose>0 0 0.06 0 -0 0</pose>
  <scale>1 1 1</scale>
  <link name='line_link'>
    <pose>0 0 0.06 0 -0 0</pose>
    <velocity>0 0 0 0 -0 0</velocity>
    <acceleration>0 0 0 0 -0 0</acceleration>
    <wrench>0 0 0 0 -0 0</wrench>
  </link>
</model>

<model name='black_line_clone'>
  <pose>0 2.33333 0.06 0 -0 0</pose>
  <scale>1 1 1</scale>
  <link name='line_link'>
    <pose>0 2.33333 0.06 0 -0 0</pose>
    <velocity>0 0 0 0 -0 0</velocity>
    <acceleration>0 0 0 0 -0 0</acceleration>
    <wrench>0 0 0 0 -0 0</wrench>
  </link>
</model>

<model name='black_line_clone_0'>
  <pose>0 4.666 0.06 0 -0 0</pose>
  <scale>1 1 1</scale>
  ...
</model>
```

Additionally, QR code tags were placed at key locations, such as loading/unloading points, stations, and before turns, to provide the robot with precise location information.

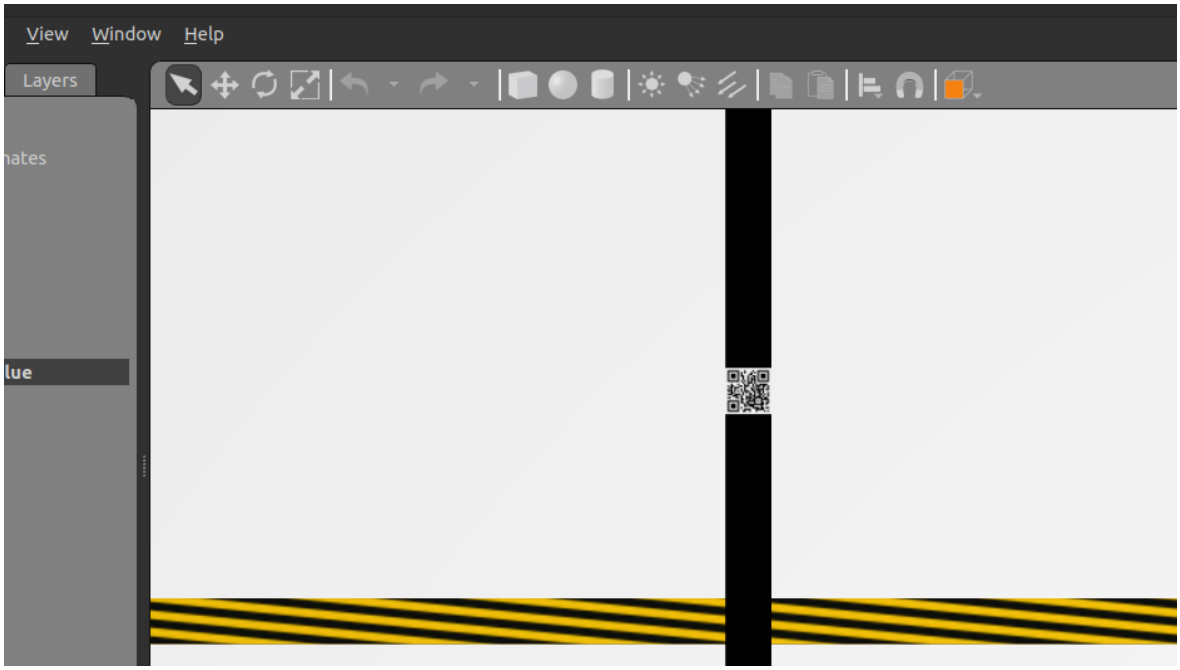


Figure 4.86: Example of Qr Code tag before a loading point

the `<friction>` tag parameters of the white floor, displayed in fig. 4.85, were specifically chosen to replicate the conditions of wood flooring. The static friction coefficient was set to 0.4, representing the resistance to the start of sliding motion. The dynamic friction coefficient was set to 0.35, modeling the friction while the robot is already sliding. These values were carefully selected to ensure that the robot's interactions with the floor behave as expected in real-world conditions.

The `<static>` tag ensures that the floor model is fixed and does not move during the simulation. The `<pose>` tag is used to position the floor in the environment, placing it just slightly above the ground to avoid collision with the ground plane. The `<collision>` tag defines the physical properties for detecting interactions, including the geometry of the floor as a box shape. Additionally, the `<visual>` tag defines how the floor appears in the simulation, with the material set to a white color to represent the wooden floor's appearance.

Code Block 4.3.37: wooden floor and a platform with collision and inertia.

```
<model name='wooden_floor'>
<static>1</static>
<pose>0 0 0.01 0 -0 0</pose>
<link name='floor_link'>
  <collision name='floor_collision'>
    <geometry>
      <box>
        <size>12 17 0.1</size>
      </box>
    </geometry>
    <max_contacts>10</max_contacts>
    <surface>
      <contact>
        <ode/>
      </contact>
      <bounce/>
      <friction>
        <ode/>
        <torsional>
          <ode/>
        </torsional>
        <static_friction>0.4</static_friction> <!-- Adjusted for
        ↪ wooden floor -->
        <dynamic_friction>0.35</dynamic_friction> <!-- Adjusted for
        ↪ wooden floor -->
      </friction>
    </surface>
  </collision>
  <visual name='floor_visual'>
    <geometry>
      <box>
        <size>12 17 0.1</size>
      </box>
    </geometry>
    <material>
      <ambient>1 1 1 1</ambient>
      <diffuse>1 1 1 1</diffuse>
    </material>
  </visual>
```

```
      <self_collide>0</self_collide>
      <enable_wind>0</enable_wind>
      <kinematic>0</kinematic>
    </link>
  </model>
```

4.3.21.1.1.2 Obstacles and Load Platforms:

The environment includes various *obstacles* such as walls and dynamic objects, designed to challenge the robot's obstacle avoidance and path planning abilities. *Platforms for load dynamics* were added to simulate loading and unloading points, where the robot needs to deliver or pick up objects, mimicking the tasks required during the competition.

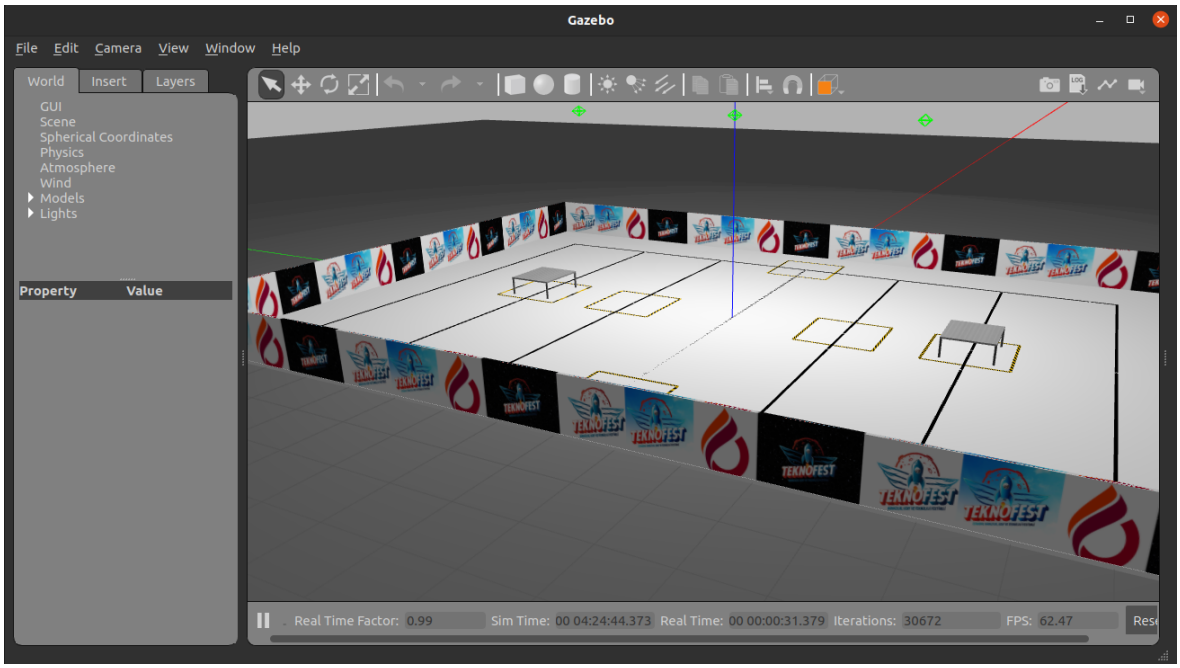


Figure 4.87: Competition aera bounded by ad hoardings

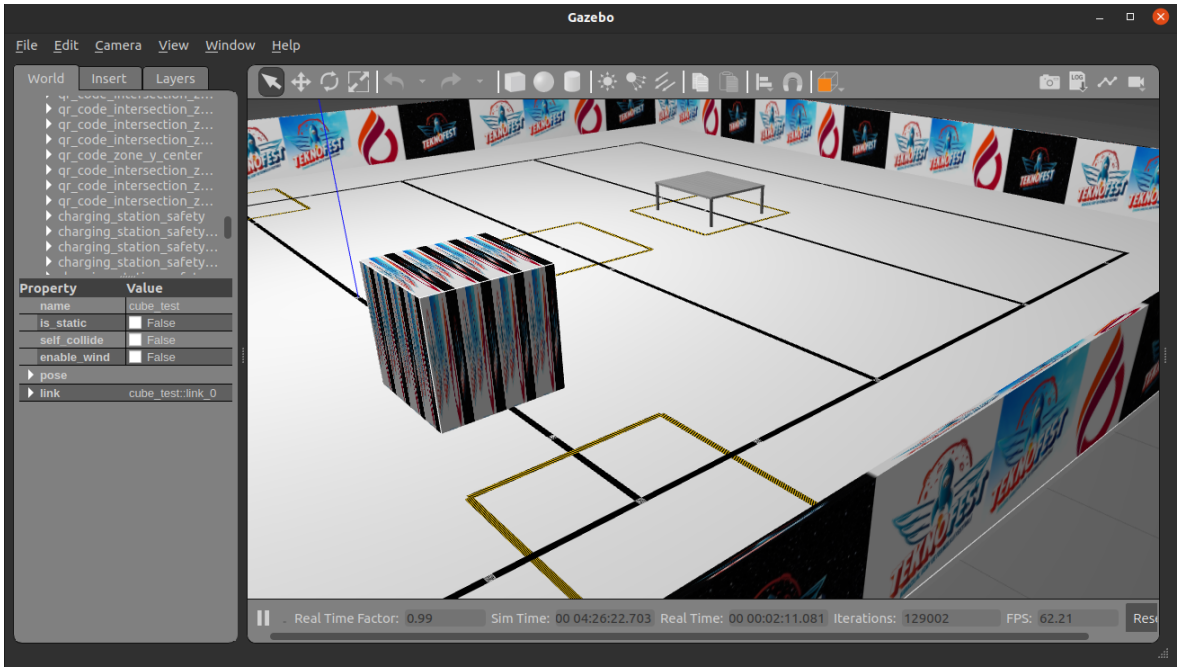


Figure 4.88: Competition area line interrupted by permanent obstacle

The platform model, as shown in the code snippet, is designed with a central table surface and four supporting legs. The structure is composed of multiple links, where each leg is defined separately, and they are all connected to the table surface through fixed joints.

The `<model>` tag defines the platform, which includes the physical properties and geometric shapes of the table and legs.
Each link, such as `table_surface` and `front_left_leg`, `front_right_leg`, etc., has its own inertial properties (`<inertial>`) and a collision geometry defined under the `<collision>` tag.

For instance, the `<collision>` tag for the table surface has a defined friction property, with a coefficient of 1 for both static and dynamic friction, simulating a high-friction surface. This friction coefficient can be adjusted to simulate the specific characteristics of a material, such as wood or metal.

Each leg is represented as a cylinder, with its own collision properties and friction settings, ensuring realistic interaction with the robot during the simulation. The friction properties of each leg are defined in the `<surface> <friction>` section using `ode` settings. These settings also include the `<bounce>` and `<contact>` properties, which help simulate realistic physical interactions between the objects in the environment. Each of these legs also has a specific pose and position in the simulation.

The joints, such as `front_left_joint`, connect each leg to the table surface with a `fixed` type, meaning that the legs do not move relative to the table. The pose of each joint defines the exact position and orientation of the legs in the simulation environment.

The overall pose of the platform, including its position in the world, is defined at the end of the model, specifying the coordinates and orientation for the platform's placement in the simulation.

Additionally, the platform's visual properties are set using the `<visual>` tag, with a custom material representing the surface texture, such as stainless steel, applied to the table's surface. To ensure accurate physical interactions, the `<inertial>` tag defines the mass and moments of inertia of the platform and its legs, ensuring a realistic response to forces and torques applied during the simulation. The `<collision>` tag specifies the geometric shape used for physics calculations, which may be simplified compared to the visual model to improve computational efficiency while maintaining accurate contact and collision responses. The platform's legs also incorporate both `<collision>` and `<inertial>` properties, allowing them to contribute to the overall stability of the structure while ensuring that the physics engine correctly simulates interactions such as impacts, weight distribution, and external forces acting on the table.

Code Block 4.3.38: Defines the platform's parameters.

```
<model name='platform'>
  <link name='table_surface'>
    <inertial>
      <mass>13.98</mass>
      <inertia>
        <ixx>0.1</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyy>0.1</iyy>
        <iyz>0</iyz>
        <izz>0.1</izz>
      </inertia>
      <pose>0 0 0 0 -0 0</pose>
    </inertial>
    <collision name='surface_collision'>
      <pose>0 0 0.367 0 -0 0</pose>
      <geometry>
        <box>
          <size>1.1 0.95 0.03</size>
        </box>
      </geometry>
      <surface>
        <friction>
```

```
        <ode>
          <mu>1</mu>
          <mu2>1</mu2>
        </ode>
        <torsional>
          <ode/>
        </torsional>
      </friction>
      <contact>
        <ode/>
      </contact>
      <bounce/>
    </surface>
    <max_contacts>10</max_contacts>
  </collision>
  ...
```

Code Block 4.3.39: Applies a stainless steel texture to the platform's surface.

```
  ...
  <visual name='surface_visual'>
    <pose>0 0 0.367 0 -0 0</pose>
    <geometry>
      <box>
        <size>1.1 0.95 0.03</size>
      </box>
    </geometry>
    <material>
      <script>
        <uri>model://platform/materials/scripts</uri>
        <uri>model://platform/materials/textures</uri>
        <name>stainless_steel_texture</name>
      </script>
    </material>
  </visual>
  ...
```

Code Block 4.3.40: Constructs the front legs.

```
...
<link name='front_left_leg'>
  <inertial>
    <mass>2.796</mass>
    <inertia>
      <ixx>0.01</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>0.01</iyy>
      <iyz>0</iyz>
      <izz>0.01</izz>
    </inertia>
    <pose>0 0 0 0 -0 0</pose>
  </inertial>
  <collision name='collision'>
    <pose>0.53 0.455 0.1835 0 -0 0</pose>
    <geometry>
      <cylinder>
        <radius>0.02</radius>
        <length>0.367</length>
      </cylinder>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>1</mu>
          <mu2>1</mu2>
        </ode>
        <torsional>
          <ode/>
        </torsional>
      </friction>
      <contact>
        <ode>
          <kp>100000</kp>
          <kd>1</kd>
        </ode>
      </contact>
      <bounce/>
```

```
      </surface>
      <max_contacts>10</max_contacts>
    </collision>
    <visual name='visual'>
      <pose>0.53 0.455 0.1835 0 -0 0</pose>
      <geometry>
        <cylinder>
          <radius>0.02</radius>
          <length>0.367</length>
        </cylinder>
      </geometry>
      <material>
        <script>
          <uri>file://media/materials/scripts/gazebo.material</uri>
          <name>Gazebo/Grey</name>
        </script>
      </material>
    </visual>
    <self_collide>0</self_collide>
    <enable_wind>0</enable_wind>
    <kinematic>0</kinematic>
  </link>
  <link name='front_right_leg'>
    <inertial>
      <mass>2.796</mass>
      <inertia>
        <ixx>0.01</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
    ...
```

Ad hoardings and obstacles were incorporated into the environment using a simple yet effective approach. A cube shape was used to create the basic geometry of these objects, with their dimensions adjusted to meet the specific requirements of the simulation. Textures were applied to the cubes to give them the appearance of real-world hoardings and obstacles, ensuring that they serve as realistic visual and physical barriers within the simulation.

These objects were assigned physical properties, such as mass and friction, and collisions were defined to ensure proper interaction with the robot. The hoardings and permanent obstacles were defined as static objects to prevent them from moving during the simulation, ensuring they remain fixed in place as intended.

The following lines provide an example of the visual and collision properties for the hoardings. These lines demonstrate how the hoardings models were assigned the name `cube_test` during the individual SDF design phase of the world. The code defines the geometry, textures, and physical properties for the hoardings.

Code Block 4.3.41: test model with physical properties.

```
...
model name='cube_test_clone_clone'>
  <link name='link_0'>
    <inertial>
      <mass>1</mass>
      <inertia>
        <ixx>0.166667</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyy>0.166667</iyy>
        <iyz>0</iyz>
        <izz>0.166667</izz>
      </inertia>
      <pose>0 0 0 0 -0 0</pose>
    </inertial>
    <pose>-0 -0 0 0 -0 0</pose>
    <visual name='visual'>
      <pose>0 0 0 0 -0 0</pose>
      <geometry>
        <box>
          <size>0.05 16.9 1</size>
        </box>
      </geometry>
      <material>
        <lighting>1</lighting>
        <script>
          <uri>model://cube_test/materials/scripts</uri>
          <uri>model://cube_test/materials/textures</uri>
          <name>teknofest_logo</name>
        </script>
        <shader type='pixel'/>
      </material>
      <transparency>0</transparency>
      <cast_shadows>1</cast_shadows>
```

```
</visual>
<collision name='collision'>
  <laser_retro>0</laser_retro>
  <max_contacts>10</max_contacts>
  <pose>0 0 0 0 -0 0</pose>
  <geometry>
    <box>
      <size>0.05 16.9 1</size>
    </box>
  </geometry>
  <surface>
    <friction>
      <ode>
        <mu>1</mu>
        <mu2>1</mu2>
        <fdir1>0 0 0</fdir1>
        <slip1>0</slip1>
        <slip2>0</slip2>
      </ode>
      <torsional>
        <coefficient>1</coefficient>
        <patch_radius>0</patch_radius>
        <surface_radius>0</surface_radius>
        <use_patch_radius>1</use_patch_radius>
        <ode>
          <slip>0</slip>
        </ode>
      </torsional>
    </friction>
    <bounce>
      <restitution_coefficient>0</restitution_coefficient>
      <threshold>1e+06</threshold>
    </bounce>
    <contact>
      <collide_without_contact>0</collide_without_contact>
      <collide_without_contact_bitmask>1</collide_without_cont
       ↪  act_bitmask>
      <collide_bitmask>1</collide_bitmask>
      <ode>
        <soft_cfm>0</soft_cfm>
```

```
            <soft_erp>0.2</soft_erp>
            <kp>1e+13</kp>
            <kd>1</kd>
            <max_vel>0.01</max_vel>
            <min_depth>0</min_depth>
          </ode>
          <bullet>
            <split_impulse>1</split_impulse>
            <split_impulse_penetration_threshold>-0.01</split_impu
             ↪ lse_penetration_threshold>
            <soft_cfm>0</soft_cfm>
            <soft_erp>0.2</soft_erp>
            <kp>1e+13</kp>
            <kd>1</kd>
          </bullet>
        </contact>
      </surface>
    </collision>
    <self_collide>0</self_collide>
    <enable_wind>0</enable_wind>
    <kinematic>0</kinematic>
  </link>
  <static>1</static>
  <allow_auto_disable>1</allow_auto_disable>
  <pose>-6.06431 0.045196 0.46 0 -0 0</pose>
  <enable_wind>0</enable_wind>
  </model>
  ...
```

The design of the QR codes followed a similar approach to most other models in the world. Each QR code was represented by a textured cube, with textures made to resemble the unique images that contain the right location information.

Code Block 4.3.42: Adds a QR code model as a thin box with a custom material.

```
<?xml version='1.0'?>
<sdf version='1.7'>
  <model name='qr_code'>
    <link name='link_0'>
      <pose>0 0 0 0 0 0</pose>
      <visual name='visual'>
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <box>
            <size>0.05 0.05 0.001</size>  <!-- Increase size of
             ↪  the box -->
          </box>
        </geometry>
        <material>
          <lighting>1</lighting>
          <script>
            <uri>model://qr_code/materials/scripts</uri>
            <uri>model://qr_code/materials/textures</uri>
            <name>qr_code_content</name>
          </script>
          <shader type='pixel'/>
        </material>
        <transparency>0</transparency>
        <cast_shadows>1</cast_shadows>
      </visual>
    </link>
    <static>1</static>
    <allow_auto_disable>1</allow_auto_disable>
  </model>
</sdf>
```

The code block 4.3.42 represents the individual model of the QR code. It contains only visual information and does not define any physical properties, as the QR code is used for location identification within the simulation. The model is defined as static to prevent movement and includes a texture that simulates the unique QR code image.

To simplify the referencing of points on the map in fig. 4.84, the area was divided into several well-defined zones. This subdivision facilitates the robot's movement management by allowing specific tasks to be assigned to each zone. All tags were then assigned unique values based on their relative locations to critical points such as loading/unloading stations, intersections, and key passage points. These QR codes serve as essential reference markers, enabling the robot's localization system to determine its exact position and adjust its trajectory accordingly.

- *Station 1:* Located between corners 1 and 4, this zone represents station 1.
- *Station 2:* Located between corners 2 and 3, this zone corresponds to station 2.
- *Zone A:* Contains the loading/unloading point A.
- *Zone B:* Contains the loading/unloading point B.
- *Zone C:* Contains the loading/unloading point C.
- *Zone D:* Contains the loading/unloading point D.

The QR code's content is specified using a `material` element in the SDF model. The `qr_code_contents.material` file defines the texture that represents the unique image of the QR code. The `material` file is referenced within the visual element of the QR code model using a `<script>` tag, which links to the texture folder containing the QR code images.

By separating the texture definition into the `qr_code_contents.material` file, it allows for easy updates and maintenance of the QR codes used in the simulation without needing to modify the main model files.

Code Block 4.3.43: Assigns materials to different intersection areas in the environment.

```
...
{
  texture_unit { texture intersection_zone_c_station_1.png }
}
}

material intersection_zone_c_station_2 {
  technique {
    pass {
      texture_unit { texture intersection_zone_c_station_2.png }
    }
  }
}

material intersection_zone_d_station_1 {
  technique {
    pass {
      texture_unit { texture intersection_zone_d_station_1.png }
    }
  }
}

material intersection_zone_d_station_2 {
  technique {
    pass {
      texture_unit { texture intersection_zone_d_station_2.png }
    }
  }
}

material intersection_zone_x_station_1 {
  technique {
    pass {
      texture_unit { texture intersection_zone_x_station_1.png }
    }
  }
}

material intersection_zone_x_station_2 {
```

```
  technique {
    pass {
      texture_unit { texture intersection_zone_x_station_2.png }
    }
  }
}

material intersection_zone_y_station_1 {
  technique {
    pass {
      texture_unit { texture intersection_zone_y_station_1.png }
    }
  }
}

material intersection_zone_y_station_2 {
  technique {
    pass {
      texture_unit { texture intersection_zone_y_station_2.png }
    }
  }
}
...
```

Figure 4.89: Qr code tag placed at the intersection of line of the zone C and the station 2

4.3.21.1.1.3 Random Clutter Generation:

Procedural generation methods were used to add *random clutter*, simulating items or obstacles that may appear unpredictably, requiring the robot to adapt to ever-changing environments.

4.3.21.1.1.4 Real-World Simulation Features:

External environmental factors such as wind were not simulated, as the competition setting assumes an indoor environment. Basic ambient and directional lighting were configured to ensure consistent visibility for sensor-based perception tasks.

Poor lighting similar to conditions in fig. 4.90 can introduce challenges such as motion blur, sensor noise, and shadow distortions, making it difficult for cameras to differentiate objects from the background. Additionally, extreme lighting conditions, such as glare or overexposure, may lead to incorrect sensor readings, affecting navigation and localization accuracy.

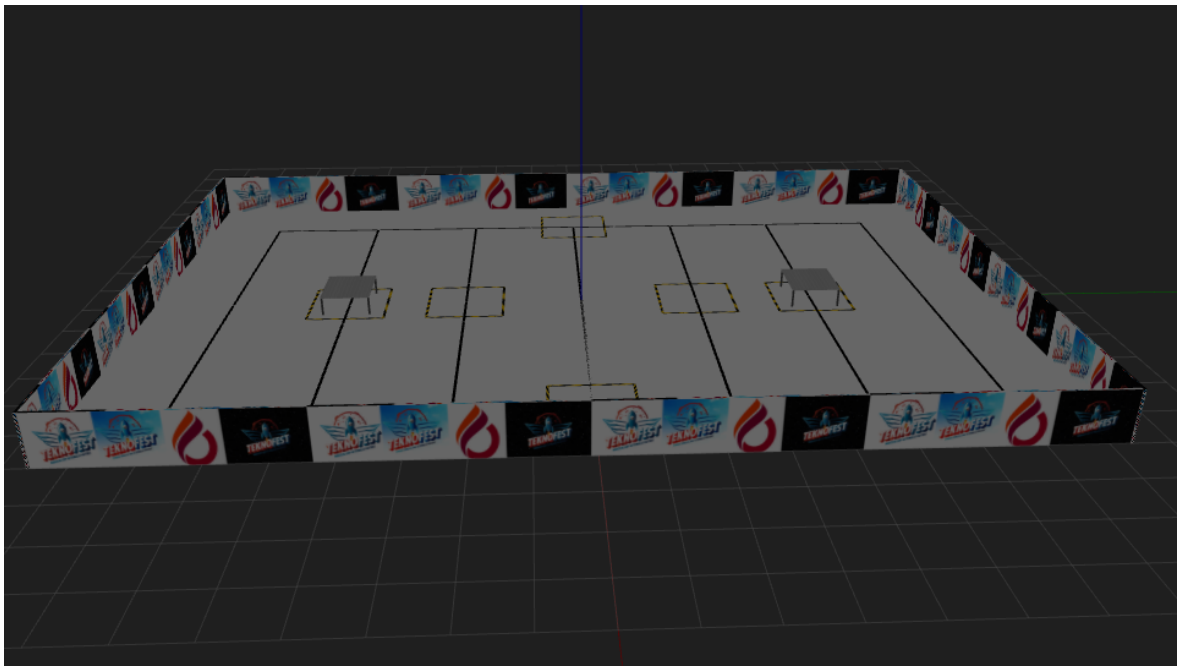


Figure 4.90: Effect of lighting conditions in simulation environment

On the other hand, good lighting shown in fig. 4.91 improves the performance of vision-based tasks such as object detection and QR code recognition. A well-lit environment with uniform brightness and minimal shadows ensures that cameras and sensors receive clear and consistent data, reducing the chances of misinterpretation due to varying illumination levels.

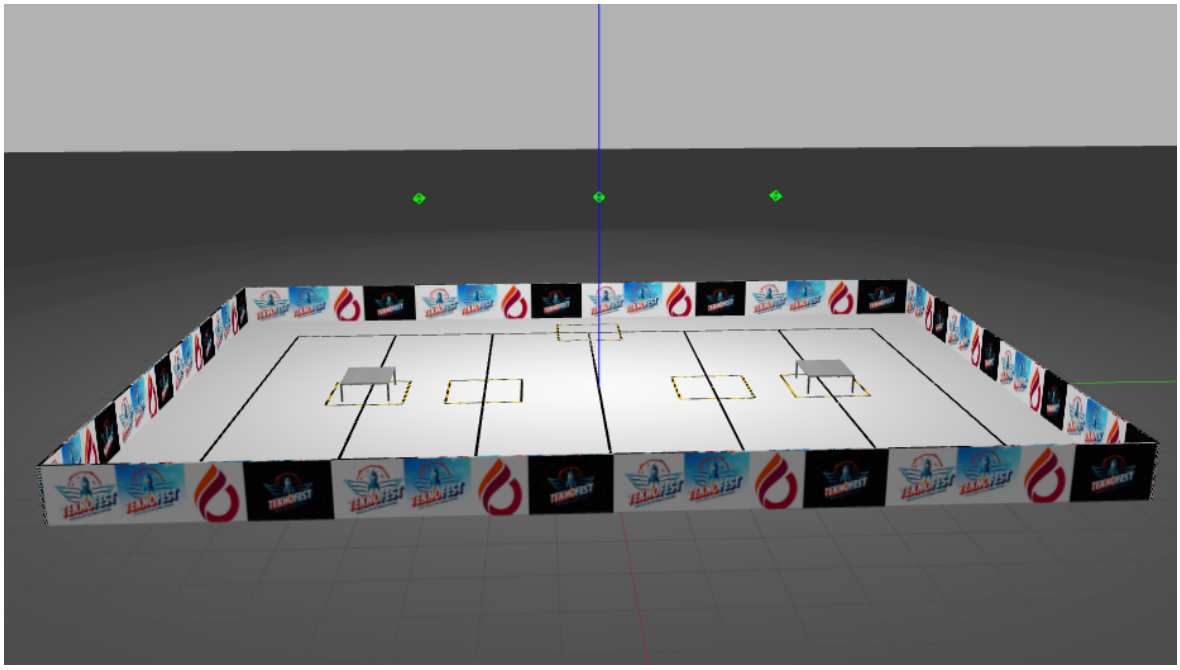


Figure 4.91: Effect of lighting conditions in simulation environment

4.3.21.2 Control and sensor Integration

in order to simulate the sensors reading from an outside world inside a simulation environment, Gazebo offers limited plugins for ROS that we can use such as: Camera - Multi-camera - Depth Camera - GPU Laser - Laser - IMU - and more. you can find more Gazebo plugins in https://classic.gazebosim.org/tutorials , here is an example of how to integrate a sensor in the simulation environment:

Code Block 4.3.44: Sensor integration in URDF

```
<robot>
  ... robot description ...
  <link name="sensor_link">
    ... link description ...
  </link>
  <gazebo reference="sensor_link">
    <sensor type="camera" name="camera1">
      ... sensor parameters ...
      <plugin name="camera_controller"
       ↪  filename="libgazebo_ros_camera.so">
        ... plugin parameters ..
      </plugin>
    </sensor>
  </gazebo>
</robot>
```

the <gazebo> tag is added to the URDF in order to initiae the pliginz between ROS and Gazebo. its worth mentioening that the gazebo package should be installed on ROS if not the Gazebo tag won be recognised by ROS, for noetic distribution (code block 4.3.45)

Code Block 4.3.45: Gazebo ROS packages installation

```
# Install Gazebo ROS Packages (includes sensor plugins and control
 ↪  integration):
sudo apt install ros-noetic-gazebo-ros-pkgs
 ↪  ros-noetic-gazebo-ros-control
# Install Robot State and Joint State Publishers:
sudo apt install ros-noetic-robot-state-publisher
 ↪  ros-noetic-joint-state-publisher
# Install ROS Control (if you plan to simulate
 ↪  controllers/actuators):
sudo apt install ros-noetic-ros-control
```

beside the sensors gazebo also provide plugins to control the actuators like *Differential Drive* but duo to limitation of number of actuators that gazebo plugins supports there is another method and that is writing a ymal file forr the simulation. in this way by publishing geometry message on the joint we can control the robot in the simulation environment.(check section 4.3.17.1):

Code Block 4.3.46: Controller configuration file

```
# Define a controller for controlling a joint by position.
# You can give any name to the controller. In this example, we call it
 ↪  "position_joint_controller".
position_joint_controller:
# Specify the type of controller.
# Here, we use the JointPositionController from the
 ↪  position_controllers package,
# which commands a joint based on the desired position.
type: position_controllers/JointPositionController

# Specify the joint this controller will manage.
# Replace 'platform_lift_joint' with the exact name of your joint as
 ↪  defined in your robot's URDF or SDF.
joint: platform_lift_joint

# Configure the PID gains for the controller.
# 'p' is the proportional gain, which reacts to the current error.
# 'i' is the integral gain, which reacts to the accumulation of past
 ↪  errors.
# 'd' is the derivative gain, which reacts to the rate of change of the
 ↪  error.
pid: {p: 100.0, i: 0.01, d: 10.0}
```

the codes in code block 4.3.46 is a configuration file for the controller that will be used to control the joint in the simulation environment. it will creat a topic in this case `/position_joint_controller/command` that will be used to control the joint by publishing on the topic. we can write a script or even publish directly on the topic via bash terminal to control the joint.

Code Block 4.3.47: Publishing a position command to the joint controller

```
# Publish a position command to the joint controller topic
rostopic pub /position_joint_controller/command std_msgs/Float64
 ↪  "data: 1.0"
```

4.4 CONTROL DESIGN AND ALGORITHM

4.4.1 Navigation

Autonomous navigation is a rapidly growing field and has become a pivotal area of research and application in robotics, with numerous applications in industries such as transportation, manufacturing, and warehousing. One of the fundamental tasks for autonomous robots is the ability to navigate their environment independently with minimal human intervention, that is a crucial factor or a key milestone for the development of intelligent systems capable of interacting with the real world. Among the various techniques used for autonomous navigation, line following remains one of the most essential and widely researched methods, especially for mobile robots, where the robot must track a specific line on the ground. While it may appear straightforward, line following involves a variety of challenges, including sensor calibration, precise control of the robot's motors, and handling of unpredictable environmental factors.For this project the primary objective was eventually designing and developing a simulated robot capable of accomplishing a specific task while following a line and avoiding obstacles and maintain accurate control over its movements within a simulated environment using the **Robot Operating System (ROS)**, which is widely used in both research and industry due to its flexibility, modularity, and wide range of libraries and tools. ROS provides a powerful framework for developing and simulating robotic applications.

Methods used to design:

It is necessary outline the methodology used to design the robot, its navigation system, and the simulation environment. The method should cover both the hardware (even though it's simulated) and software design, as well as the control algorithms used to ensure the robot follows the line autonomously.

The design of the autonomous line-following robot involved a series of methodical steps to ensure the successful integration of hardware (simulated), software (ROS-based), and control algorithms to enable autonomous navigation. The following sections describe the approach taken to design and implement the robot and its navigation system in the simulation environment.

4.4.2 For Line Detection and Following:

4.4.2.1 Camera-based Vision Method:

Using **cameras** for line-following is an advanced approach in robotics, where the robot uses visual information to detect and track a line or path. This technique, often referred to as

vision-based line following, involves utilizing computer vision algorithms to process images captured by the robot's camera and determine the robot's position relative to the line. To use a camera for line-following, the camera is needed to be mounted on the robot, typically facing downward or slightly tilted to capture the surface on which the line is drawn which is the case with the robot used for this project.

This process was accomplished through many steps via simulation such :

Setting up the Simulation Environment

The simulation platform was set as with the following components:

- **Robot Model**: The robot in the simulation should have a camera mounted on it. The robot model could be a differential-drive robot, a mobile platform with wheels, or a more complex robot with additional sensors. **The robot model used is a differential-drive mobile robot.**
- **Camera Sensor**: Simulated cameras in the environment will capture images of the ground, where the line is located. The camera was set up to view downward, directly in front of the robot.
- **Line (Path)**: lines on the ground can be created in the simulation. The line can be straight, curved, or even follow a more complex path. For this project black lines were drawn within the world created in Gazebo.
- **Simulator**: The simulation platform used is:

  Gazebo: Commonly used with **ROS** (Robot Operating System) for robot control and visualization.

Image Processing for Line Detection

After setting up the simulated robot and camera, the robot will need to process the images captured by the camera to detect the line. The core part of this system involves **image processing** and **vision algorithms**.

- **Steps for Image Processing:**
- **Capture Image from the Camera**:

  In the simulation, it is necessary to get access to the camera feed in real-time. With **ROS**, there can be subscriber to the camera's image topic such **/camera/rgb/image_raw**

- **Convert the Image to Grayscale**:

  grayscale conversion can be used to reduce the complexity of the image. Since the line will typically have a contrasting colour (black or white), working with a grayscale image makes it easier to detect the line.

- **Thresholding** will convert the grayscale image into a binary image where the line is white (or black) and the background is the opposite colour. You can use a simple threshold or adaptive thresholding for better results in varying lighting conditions.

- **Detect the Line**:

  Once the image is binary, **edge detection**, **contour detection**, or **Hough Transform** can be used cto detect the line. For instance, you can use **Hough Line Transform** to detect straight lines. For the project they were combined and used together for line detection.

- **Determine the Robot's Position Relative to the Line**:

  The centroid of a detected line is calculated t hoo measure or estimate w far the robot is from the centre of the line in the camera frame.

4.4.2.2 Control Algorithm for Line Following

Once the line is detected, the robot needs to be controlled to follow it. The basic principle is to adjust the robot's steering based on its position relative to the line. PID Controller algorithm can be used to keep the robot following line in a steady way.

The **PID (Proportional-Integral-Derivative)** controller is commonly used in line-following robots.

- **Proportional**: The steering correction is proportional to how far the robot is from the line's centre.

- **Integral**: This helps eliminate accumulated errors over time.

- **Derivative**: This predicts and reacts to the rate of change in the error (speed of deviation).

4.4.2.3 Integrating the Line Following with the Simulator

Finally, the image processing and control algorithms needed to be integrated into the simulation environment. With ROS this integration is usually made through Gazebo. When **ROS** is used the robot cab ne set up with a simulated camera and subscribe to the camera feed. ROS provides libraries like CV BRIDGE to convert the ROS image messages into

4.4.3 For Building Map

4.4.3.1 LIDAR

The map was built using SLAM (Simultaneous Localization and Mapping) in a simulation thanks to the lidar sensor. Using LIDAR (Light Detection and Ranging) with SLAM (Simultaneous Localization and Mapping) is a common and effective approach for building a map of an environment while simultaneously localizing the robot within that environment. In ROS, LIDAR is often used as the primary sensor for SLAM, especially for 2D SLAM algorithms like GMapping and Hector SLAM. The algorithm used for SLAM was GMapping in this project.The lidar can be used to avoid obstacles as well.

4.4.3.2 QR Code Scanning

QR Code Detection with OpenCV was used, the OpenCV's QRCodeDetector detects and decode QR codes in the robot's camera feed. The QR code are used to help the robot to navigate autonomously, the QR code information to know its current position and calculate the closest path to the destination.

After building the map , it will be saved for autonomous navigation purpose.

4.4.4 Computational-power-and-resources

4.4.4.1 cpu-and-gpu-limitations

- *CPU Limitations*: Simulations, especially those involving complex algorithms like SLAM or sensor fusion, can be CPU-intensive. The computational demand increases with the complexity of the robot's tasks, such as real-time mapping, localization, or decision-making. A limited CPU performance could cause delays, lag, or even make real-time processing difficult.

- *GPU Constraints*: If you're using computer vision algorithms (e.g., QR code recognition, camera-based line-following), a GPU can greatly accelerate image processing. However, not all simulations leverage GPU power, and if the GPU is not sufficient or not utilized properly, the simulation could run slower or experience bottlenecks.

4.4.4.2 memory-ram-constraints

- *High Memory Usage*: Simulations that involve large environments, dense sensor data (such as LIDAR point clouds), or high-resolution images can require a significant amount of RAM. If the memory usage exceeds the available capacity, it can lead to slow performance, crashes, or even the inability to run the simulation at all.

- *Data Storage for Sensor Data*: Storing large amounts of sensor data (e.g., from LIDAR, cameras, IMUs) generated during a simulation can strain the system's storage, especially if you need to log or save sensor outputs for later analysis. In a large-scale simulation, this could be a limiting factor.

4.4.4.3 real-time-performance

- *Real-Time Simulation Constraints*: Achieving real-time performance in simulations can be difficult, especially when dealing with complex tasks such as real-time SLAM or path planning. The simulation might not run at the required frame rate (e.g., 30Hz or higher), leading to delays in robot control and decision-making.
- *Simulation Time vs Real Time*: If the simulation is not running at real-time speed (1:1 with physical time), it can make it harder to validate time-sensitive behaviors. For example, a robot using SLAM may not update its map fast enough in a simulation that runs too slowly, affecting navigation and decision-making in real-world applications.

4.4.5 Complexity-of-the-simulated-environment

4.4.5.1 size-and-detail-of-the-simulation

- *Environmental Complexity*: Large and complex simulated environments with many dynamic obstacles, detailed textures, and various types of surfaces can be computationally expensive. More detailed environments require more processing power to render, simulate physics, and handle sensor data.
- *Realistic Physics*: Physics engines in simulations (e.g., Gazebo, V-REP) simulate the interactions between the robot and the environment, including forces like friction, gravity, and object collisions. While necessary for realistic testing, these physics simulations can be computationally demanding, especially in dynamic environments with many objects.

4.4.5.2 dynamic-objects-and-movements

- *Real-Time Object Simulation*: When simulating environments with moving objects (e.g., pedestrians or vehicles), the computation required to simulate their motion and interactions with the robot can add significant overhead. This becomes particularly challenging if multiple objects are moving in complex patterns at the same time.

4.4.6 Simulating-sensor-data

4.4.6.1 a.-sensor-data-processing-load

- *LIDAR and Point Cloud Data*: LIDAR sensors generate large amounts of data, especially in 3D environments. Processing the point cloud data in real time requires significant computing resources, particularly when applying algorithms like SLAM to build and update maps. The more data points and the larger the map, the more processing power is required.

- *Camera Data for Visual Processing*: Cameras generate high-resolution images that need to be processed for tasks like QR code detection or line-following. Processing these images (especially with advanced computer vision techniques such as feature detection or deep learning) can be computationally expensive. Depending on the image resolution and the complexity of the detection algorithms, this could put a strain on the available computing resources.

4.4.6.2 Real-time-sensor-fusion

- *Combining Data from Multiple Sensors*: If you are fusing data from multiple sensors (e.g., combining LIDAR, IMU, camera data for SLAM), the computational load increases significantly. Sensor fusion algorithms, such as Kalman Filters or particle filters, need to process data from all sensors in real time, which may not be feasible with limited computational resources.

4.4.7 Algorithmic-constraints

4.4.7.1 Computational-cost-of-slam

- *SLAM Algorithm Complexity*: Algorithms used for SLAM (like Extended Kalman Filters, GraphSLAM, or particle filters) are computationally expensive, especially when the robot has to map a large area in real-time. As the environment grows, the number of calculations needed to maintain and update the map increases, potentially leading to slower performance or the need for more powerful hardware.

- *Map Size and Resolution*: The higher the resolution and accuracy of the map, the more computational resources are required to store and update it. Large maps with high-detail features demand significant memory and processing power to handle updates, store the data, and process localization.

4.4.7.2 path planning and decision making

- *Complex Path Planning Algorithms*: Path planning algorithms, such as A*, RRT, or D* algorithms, may require substantial computational resources depending on the complexity of the environment. If your robot is navigating a large or cluttered environment, calculating collision-free paths in real-time becomes a challenge, especially if there are many dynamic obstacles to avoid.

- *Decision Making Algorithms*: When the robot makes decisions based on sensor input, running machine learning algorithms or decision trees to determine the next action (e.g., go towards a QR code or follow a line) can also be computationally expensive. Depending on the complexity of the logic, this could limit the speed and responsiveness of the simulation.

4.4.8 Simulation Software limitations

4.4.8.1 Simulation Engine Efficiency

- *Performance of the Simulation Engine*: Different simulation platforms (Gazebo, V-REP, Webots, etc.) have varying levels of efficiency. Some simulators might be highly optimized for certain types of robots or algorithms, while others may be slower or require more resources to simulate complex systems. Choosing the right simulation platform for your application is critical for balancing performance and accuracy.

- *Plugin Overhead*: Many simulators rely on plugins for additional functionality (e.g., camera sensors, LIDAR). The more plugins you add or the more complex the plugin behavior, the more computational overhead is introduced. This can affect simulation performance, especially in large-scale environments or real-time applications.

4.4.8.2 Parallelization And Multithreading

- *Limited Parallel Processinh*: Not all simulation environments or algorithms are well-optimized for parallel processing, which means that tasks may not fully utilize multi-core CPUs or GPUs. For example, in simulations involving real-time SLAM, the ability to parallelize sensor data processing or SLAM computation can significantly reduce the computational load, but not all algorithms or environments support this effectively.

- *Threading Issues*: Running multiple processes (e.g., sensor readings, actuator control, SLAM) in parallel in a simulation may introduce issues such as deadlock or race

conditions if not properly managed. This can lead to slower simulation times or unresponsive behavior in your robot model.

4.4.9 Simulating-Real-Time-Constraints

4.4.9.1 Real Time Control vs. Simulation Speed

- *Synchronization Issues*: Achieving true real-time performance is difficult in a simulation. Many simulations allow you to adjust the time scale (e.g., running the simulation faster than real time for testing), but this introduces the issue that the control algorithms might not have the same timing constraints they would in a real robot. Ensuring that the simulation runs with a fixed time step, synchronized with the robot's control loops, is essential but can be computationally demanding.

- *Hardware-in-the-loop (HIL) Simulation*: When integrating real hardware with a simulator (e.g., for testing SLAM on real sensors), the latency and real-time computation required can cause performance bottlenecks. This is especially true if communication between the simulation and physical robot hardware is slow or not synchronized properly.

The design of our industrial robot involved a structured approach that included problem formulation, engineering problem-solving, and the application of appropriate analysis and modelling methods. our primary role in the project focused on the programming tasks, ensuring the implementation of the robot's control system, QR code recognition, navigation algorithms, and communication between ROS and the web interface.

4.4.10 Problem Formulation

The key engineering challenges we addressed through programming were:

1. *Navigation System with junction detection*: Implementing a control algorithm to follow a black line and detect a junction.
2. *Load and Unload Identification*: Developing a QR code recognition system.
3. *Communication Between ROS and Web Interface*: Ensuring seamless data transfer between the robot's ROS-based system and a web-based user interface.

4.4.11 Engineering Problem-Solving

4.4.11.1 Navigation System (Line Following and junction detection Algorithm)

The robot needed to detect and follow a black line accurately. To achieve this, I explored two potential solutions:

- Computer Vision Approach: Using OpenCV to process camera input and detect the black line.
- Sensor-Based Approach: Using simulated infrared sensors in ROS.

We opted for the *computer vision approach* due to its flexibility in real-world applications. I implemented an algorithm using *OpenCV in Python,* which:

- Captured images from the robot's camera.
- Applied image processing techniques (grayscale conversion, thresholding, edge detection).
- Used contour detection to track the black line and adjust the robot's movement accordingly.

4.4.11.2 Load and Unload Point Detection (QR Code Scanning)

To recognize loading and unloading points, I implemented a *QR code detection system* using *QReader library* for detecting and decoding QR codes.

The system was designed to extract location data from QR codes and send it to the navigation system to adjust the robot's path.

4.4.11.3 Communication Between ROS and Web Interface

Since ROS does not natively communicate with web technologies, I implemented a *real-time communication system* using a *WebSockets (Socket.IO)* to enable bidirectional data transfer and *ROSBridge* to translate ROS messages into JSON format for the web interface.

This setup allowed users to monitor the robot's status and send commands through a web-based interface built with *React.js.*

4.4.11.4 Application of Theoretical and Applied Knowledge

The work applied key theoretical and practical knowledge across multiple domains. Computer vision and image processing techniques were utilized for line detection and QR code recognition, ensuring accurate navigation and object identification. Robotics and control algorithms were implemented to facilitate smooth and efficient movement. Networking and web communication played a crucial role in establishing real-time data transfer between ROS and the web interface, enabling seamless interaction. Additionally, software engineering best practices were followed to structure ROS nodes effectively, optimize system performance, and debug potential issues, ensuring a well-organized and robust implementation.

Through this structured approach, I successfully developed the necessary programming solutions for the industrial robot, ensuring accurate navigation, QR code recognition, and seamless communication between the robot and the web interface.

4.4.12 User Interface

The UI is structured into modular React components, each dedicated to displaying specific robot-related information. These components dynamically consume state from the *Zustand core*, ensuring real-time synchronization between the ROS updates and UI rendering. This modular approach enhances maintainability and flexibility, allowing for seamless updates to the user interface as data changes, while also ensuring the UI remains responsive and in sync with the backend processes.

The system enables real-time robot movement control with built-in process validation, ensuring smooth operation during task execution. It supports live camera and QR code feed display, providing real-time visual feedback. Dynamic map updates are implemented based

on the robot's movement and scanned locations, offering continuous situational awareness. Task execution and monitoring are integrated, including cargo loading and unloading, with clear updates on progress. Interactive gear control is featured, with process-state validation to prevent gear changes during active tasks, ensuring safe and efficient operation throughout the system.

4.4.12.1 Design Hierarchy

The application follows a structured GUI hierarchy where the *App* component serves as the central entry point, interacting with a *Core Store* for managing state. Within the App, the *MainLayout* organizes key sections, including a *NavBar*, a *Task* section, and a *Remote* module. The *Remote* module further breaks down into essential components such as *Camera, Map, RemoteButton,* and *Gear & Speed*, suggesting it handles functionalities related to remote control and navigation. This modular design ensures a clear separation of concerns, making the application easier to maintain and extend. The overall structure is visually represented in fig. 4.92 below.

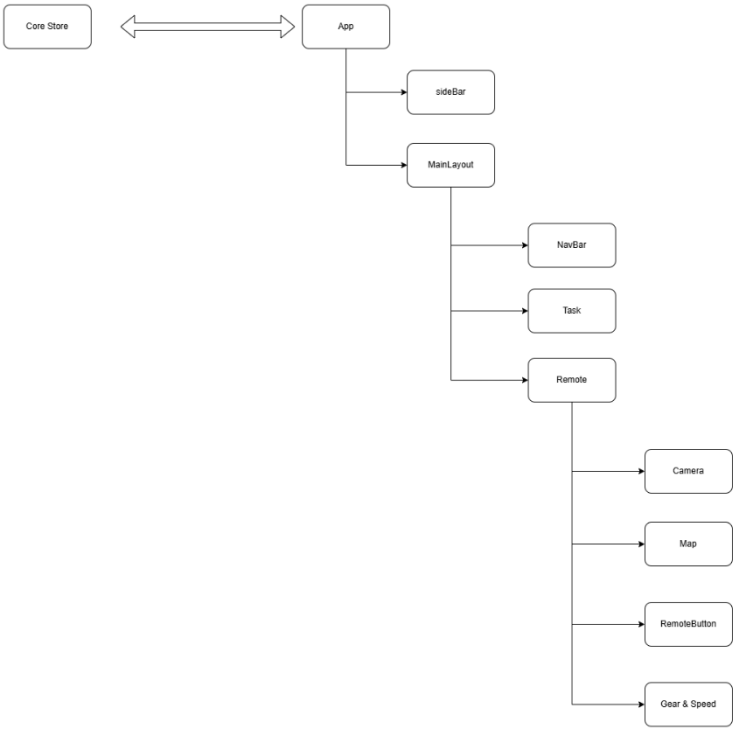


Figure 4.92: GUI hierarchy

4.4.12.2 Output

The graphical user interface of the robot, segmented into several functional areas for ease of operation:

1. *Menu*: On the left side, we have a straightforward menu with two options: ”Remote Control” and ”Task.” This menu allows users to navigate between different functionalities of the system easily (see fig. 4.93).

2. *Remote Control Tab*: This section is the heart of the interface and is further divided into three parts (see fig. 4.94):

    - *Main Camera View*: Dominating the top centre is a large grey box labelled ”Main Camera.” This is likely where the live feed from the camera appears, giving users a visual of the controlled environment.
    - *Control Panel*: Located below the camera view, this panel includes directional buttons (*up, down, left, right*) and a central button labelled ”S” for emergency stop There's also a toggle switch marked ”A” and ”M” and a speed indicator displaying ”0.00 m/s.” These controls are probably used to manoeuvre the remote vehicle or robot.
    - *Map*: On the right side, we see a white area with a grid of blue squares, a red and green marker, likely indicating the position of the robot or points of interest. There are two additional buttons at the bottom, a blue one with a ”+” sign and a red one with a ”-” sign, possibly for zooming in and out.

3. Task Tab: designed to manage tasks and monitor activities. The interface is divided into distinct sections for better functionality (fig. 4.95):

    - *Mapping Button*: This button is used to send the mapping task to the robot.
    - *Task Display*: Under the ”Mapping” section, there's a white text box labelled ”Task display,” which displays tasks and other relevant information.
    - *Main Display Area*: The majority of the screen is occupied by a large dark area that serves as the main display or workspace for the system's operations.
    - *Send Task Button*: At the bottom of the screen, this button allows users to send tasks to the robot.

This interface offers a clear, user-friendly design for effectively managing the robot.

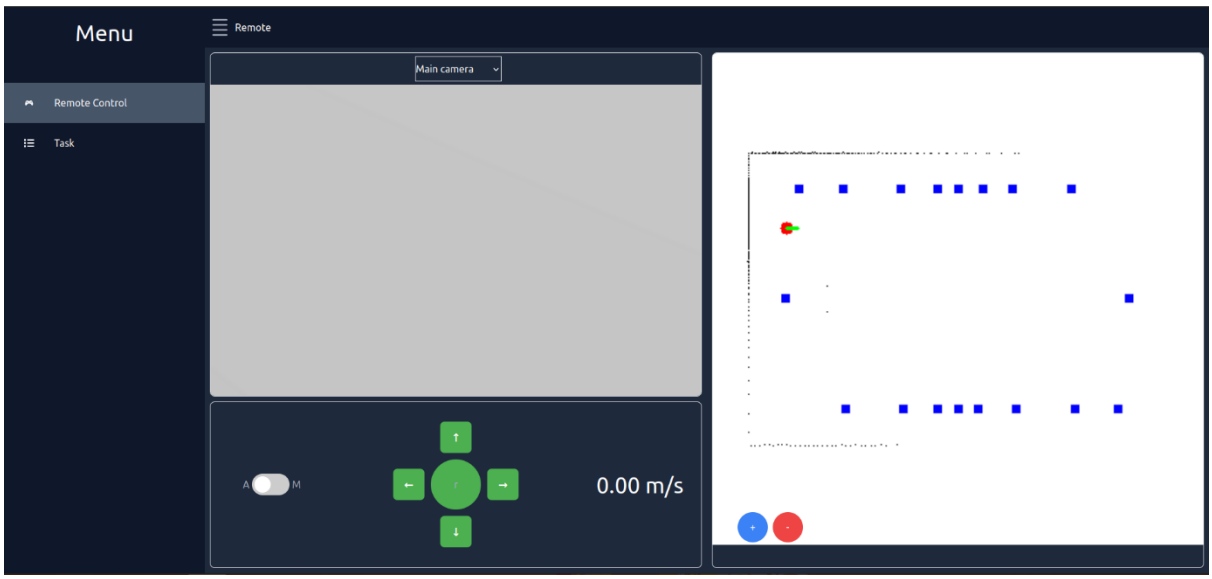


Figure 4.93: Opened menu & remote control Tab

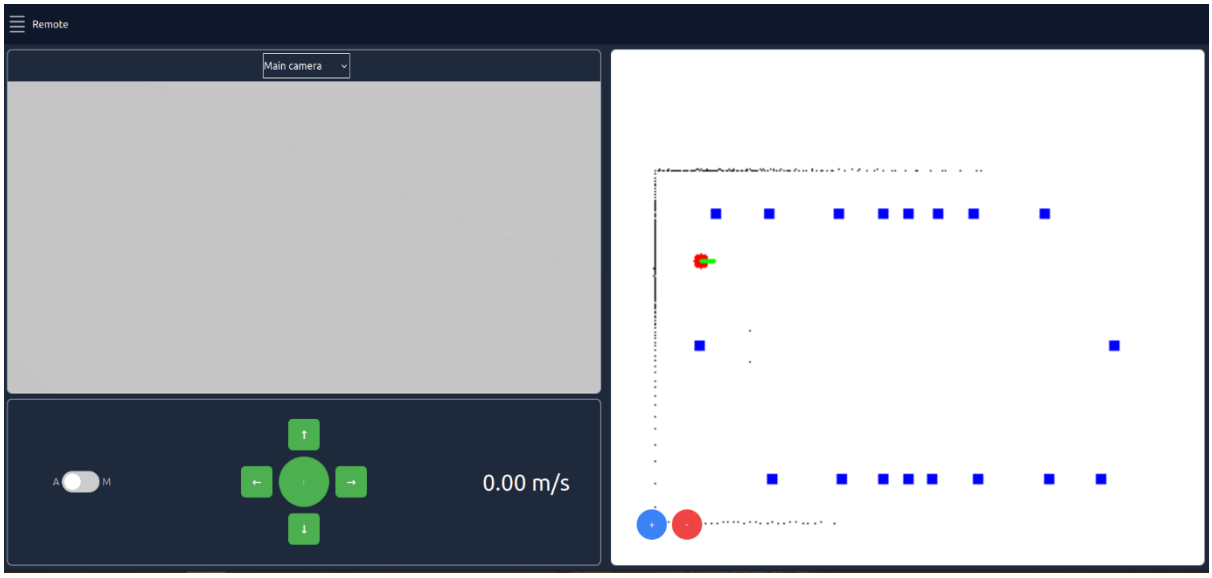


Figure 4.94: Remote control tab

Figure 4.95: Task Tab before mapping

4.4.12.3 Codes and background implementation

4.4.12.3.1 Core Store Implementation – The Most Important Element

a centralized state management system using *Zustand* to handle application state, WebSocket communication, and real-time data updates.

Key Responsibilities and Implementation Details:

State management was handled using *Zustand* by creating a global store (core, see fig. 4.92) to manage the application's state efficiently. This included tracking the connection status through *connectedSocket* and *connectedRos*, as well as managing the UI state with menu and *menuOpened*. Real-time data, such as *cameraData*, *camera_qr_data, mapData, speed, gear, zone,* and *station*, was also stored to ensure seamless updates. Additionally, the application's process state was maintained using *processState*. To facilitate state updates, setter functions like *setmenu* and *setmenuOpened* were implemented, allowing for specific property modifications.

Code Block 4.4.1: All states of core element with their default value

```
export const core = create((set,get) =>
({ connectedSocket : false,
    connectedRos : false,
    menu : "Remote",
    menuOpened : false,
    socket : null,
    cameraData : null,
```

```
    camera_qr_data : null,
    speed : 0,
    mapData : null,
    gear : 0,
    zone : null,
    station : null,}))
```

WebSocket integration was implemented using Socket.IO to enable real-time communication between the frontend and backend. The *initializeSocket* function was developed to establish a connection with the WebSocket server at `http://localhost:5000` (see fig. 4.93). It managed connection and disconnection events, ensuring the *connectedSocket* state was updated accordingly. Additionally, it listened for real-time data streams, including *camera_feed, camera_qr_feed, map_feed, speed, gear_state, task_response, mapping_output,* and *processState*, allowing for seamless data exchange and synchronization across the application.

Code Block 4.4.2: socket connection

```
const socket = io('http://localhost:5000');
set({ socket });
```

Real-time data handling was implemented to ensure seamless updates and synchronization across the application. Camera feeds were processed and stored as base64-encoded images in the state variables *cameraData* and *camera_qr_data*, enabling real-time visualization. The *mapData* state was continuously updated with real-time map images received from the backend. Additionally, the robot's speed and gear state were tracked through WebSocket events, updating the speed and gear properties accordingly. Task responses from the backend were displayed using react-hot-toast, providing instant user feedback and enhancing the overall user experience.

Code Block 4.4.3: socket connection

```
camera_feed : async function(data){
        set({cameraData : data.image})
    },
    camera_qr_feed : async function(data){
        set({camera_qr_data : data.image})
    },
    map_feed : async function(data){
```

```
        set({mapData : data.image})
    },
```

he frontend and backend. The *sendTask* function was implemented to transmit task sequences, such as loading and unloading, to the backend via WebSocket. Task data, including *task_name* and params, was structured and emitted through the *robot_task* event. For mapping, the mapping function was triggered to initiate the process by emitting a *robot_task* event with the necessary task data. Additionally, the *mapping_output* event was processed to serialize and store zone and station data in the state variables zone and station, ensuring efficient real-time updates.

Serialization of mapping data was handled through the serialize function (see fig. 4.95), which processed raw location data received from the backend. This function filtered and organized intersection and station data into structured formats, ensuring clarity and usability. The processed data was then stored in the state variables zone and station, enabling seamless integration with the UI for real-time visualization and interaction.

Code Block 4.4.4: Serializing locations efficiently

```
serialize : async function(locations){
      let lcs = [], st = []
      locations.forEach(location => {
        let nm = location.location.split("_")
        if(nm[0] == "intersection"){
          if (nm[2] == "x" || nm[2] == "y"){
          }else{
            let found = false
            for(let i = 0; i < lcs.length; i++){
              if (lcs[i].location == nm[1] + " " + nm[2]){
                found = true
              }
            }
            if (!found){
              lcs.push({
                location : nm[1] + " " + nm[2],
                x : location.x, y : location.y,
              })
            }
          }
        }else if (nm[0] == "station"){
          let found = false
```

```
            for(let i = 0; i < st.length; i++){
              if (st[i].location == nm[0] + " " + nm[1]){
                found = true
              }
            }
            if (!found){
              st.push({location : nm[0] + " " + nm[1],
              x : location.x, y : location.y,
            })
          }
        }
      });
      set({zone : lcs})
      set({station : st})
    },
```

Gear control was managed through the *changeGear* function, allowing the robot's gear state to toggle between *automatic* and *manual* modes. To ensure safe operation, gear changes were restricted while an active process was running *(processState === "Processing")*. Additionally, react-hot-toast was used to provide user feedback, notifying users when gear changes were blocked due to ongoing tasks.

Direction control was handled through the *sendDirection* function, which transmitted movement commands such as *UP, DOWN, LEFT*, and *RIGHT* to the backend via WebSocket. To maintain system integrity, movement commands were restricted during active processes *(processState === "Processing")*, preventing conflicts with ongoing tasks. User feedback was provided to ensure clarity when movement commands were blocked.

The outcome was the successful delivery of a scalable and efficient state management system that seamlessly integrates with WebSocket communication. This integration enabled real-time updates for critical data, including camera feeds, map data, speed, and gear state, significantly enhancing the application's interactivity and responsiveness. Additionally, the system provided a robust foundation for task management, mapping, and robot control, ensuring a smooth and intuitive user experience while maintaining high performance and reliability.

4.4.12.3.2 App Component

the root App component serves as the entry point for the application.

The application was enhanced with several key integrations to improve functionality and user experience. *React Router (BrowserRouter)* was integrated to enable client-side routing, allowing for seamless navigation between pages. Conditional rendering logic was added to display a loading spinner (Loader component) while waiting for the socket connection to be established, ensuring users are informed during the connection process. The layout was structured using flexbox for a responsive design, dividing the interface into a SideBar and MainLayout component for better organization. Additionally, React Hot Toast (Toaster) was integrated to provide user notifications and feedback, enhancing interactivity and responsiveness throughout the application.

The outcome:

The creation of a scalable and modular application structure, designed to support real-time communication capabilities seamlessly. By implementing loading states and real-time notifications, the application ensures a smooth user experience, keeping users informed and engaged during the connection process and throughout their interactions with the system. This structure not only enhances performance but also improves the overall user interface, making it responsive and intuitive. See ()code block 4.4.5) for implementation.

Code Block 4.4.5: App Component

```
function App() {
  const { initializeSocket, connectedSocket, connectedRos } = core();

  useEffect(() => {
    initializeSocket();
  }, [initializeSocket]);

  if (!connectedSocket) {
    return <Loarder />;
  }

  return (
    <BrowserRouter>
      <div className="flex flex-row w-full h-full">
        <SideBar />
        <MainLayout />
      </div>
      <Toaster />
```

```
    </BrowserRouter>
  );
}

export default App;
```

4.4.12.3.3 MainLayout Component

This component manages the main content area and routing logic. See code block 4.4.6 for implementation.

Key Responsibilities:

The application was structured with React Router using Routes and Route to define navigation paths. A default route (/) was set to render the Remote component, while additional routes were configured for the /remote path to display the Remote component and the /task path for the Task component. Conditional rendering was implemented to display a loading spinner (Loader component) when the socket connection is not yet established, ensuring a smooth user experience during the initial connection phase. The layout was designed to be responsive using CSS, allowing the content area to adjust dynamically based on the viewport size. Additionally, a *NavBar* component was integrated at the top of the layout to provide consistent navigation across the application, enhancing usability and accessibility.

Outcome:

Was the creation of a centralized and reusable layout structure for the application, promoting maintainability and scalability. By implementing proper loading states and routing, seamless navigation was ensured, allowing users to transition between different sections smoothly. This structure enhanced the overall user experience, providing consistent and responsive design elements across the application, while also ensuring that users are kept informed during the socket connection process.

Code Block 4.4.6: MainLayout Component

```
const MainLayout = () => {
    const { connectedSocket, menuOpened } = core();
    return (
    <div className="w-full h-full overflow-hidden">
        <NavBar />
        <div className="w-full h-[calc(100%-64px)]">
        <Routes>
            <Route path="/" element={connectedSocket ? <Remote /> :
            ↪  <Loader />} />
            <Route
            path="/remote"
            element={connectedSocket ? <Remote /> : <Loader />}
            />
            <Route path="/task" element={connectedSocket ? <Task /> :
            ↪  <Loader />} />
        </Routes>
        </div>
    </div>
    );
};
export default MainLayout;
```

4.4.12.3.4 Remote Component

Remote component serves as the central interface for remote control functionality.

The application features a dual-panel layout implemented using *CSS Flexbox*, enhancing responsiveness and user experience. The left panel integrates a live camera feed alongside a control interface equipped with directional buttons (*RemoteButton*) for navigation. A gear-switching mechanism is incorporated, utilizing a toggle switch to alternate between *automatic (A)* and *manual (M)* modes. This functionality leverages *React's useEffect* and *useRef hooks* for efficient state management. Throughout, responsive and dynamic UI updates are ensured, adapting seamlessly to user interactions and state changes. See code below for implementation.

Code Block 4.4.7: Remote control interface for navigation

```
const Remote = (props) => {
    const {sendDirection, changeGear, gear} = core()
    const gearref = React.createRef()
```

```
    const onReset = function(e){ sendDirection("STOP") }
    const switchGear = function(e){
        e.preventDefault()
        changeGear()
    }
    useEffect(() => { gearref.current.checked = gear == 1}, [gear])
return (
<div className=
 ↪  "w-full h-full overflow-hidden flex flex-row bg-slate-800">
  <section className="w-1/2 p-2">
    <div className="w-full h-2/3 flex flex-col items-center
    border rounded-lg overflow-hidden">
      <Camera/>
    </div>
    <div className="w-full h-[calc((100%/3)-8px)]
    border rounded-lg mt-2 flex flex-row">
      <div className=" w-1/4 h-full flex items-center justify-center">
        <label className="switch-label mr-1">A</label>
          <label className="switch">
            <input type="checkbox" name="switch"
            id = "switch" ref={gearref} value={gear == 1}
             ↪  onClick={switchGear}/>
              <span className="slider round"></span>
          </label>
        <label className="switch-label ml-1">M</label>
      </div>
      <div className=
       ↪  "circle-section flex items-center justify-center  w-2/4 h-full"
       ↪  >
        <div className="circle-container relative ">
          <button className="center-circle" onClick =
           ↪  {onReset}>r</button>
            <RemoteButton direction="UP"/>
            <RemoteButton direction="DOWN"/>
            <RemoteButton direction="LEFT"/>
            <RemoteButton direction="RIGHT"/>
        </div>
      </div>
      <div className=" w-1/4 h-full"><Speed/></div>
    </div>
```

```
  </section>
  <section className="w-1/2 flex items-center justify-center p-2">
    <div  className="w-full h-full border rounded-lg flex flex-col
    items-center relative overflow-hidden">
      <Map ></Map>
    </div>
            </section>
    </div>
    );
};
export default Remote;
```

4.4.12.3.5 Camera Component

The Camera component displays live camera feeds and enable camera switching functionality.

A dynamic camera feed display was implemented by processing base64-encoded image data from the backend, supporting both the Main Camera and QR Code Camera. *React's useEffect* was utilized to ensure the image source (*imgRef*) updated whenever new camera data (*cameraData* or *camera_qr_data*) was received (see code block 4.4.8). A camera selection dropdown was added to enhance user interaction, allowing users to switch between the main camera and QR code camera views. Memory management was efficiently handled by revoking object URLs when the component unmounted or the camera data changed. The component was styled using CSS to ensure the camera feed was responsive and fit seamlessly within the layout.

Code Block 4.4.8: UseEffect which update camera image each render

```
useEffect(function(){
        if(cam === "front"){
            if (cameraData) {
                const imageUrl = `${cameraData}`
                imgRef.current.src = imageUrl;

                // Clean up the object URL when component is unmounted
                return () => {
                    URL.revokeObjectURL(imageUrl);
                };
            }
        }
```

```
        else if (cam === "back"){
            if (camera_qr_data) {
                const imageUrl =
                 ↪  `${camera_qr_data}`
                imgRef.current.src = imageUrl;

                // Clean up the object URL when component is unmounted
                return () => {
                    URL.revokeObjectURL(imageUrl);
                };
            }
        }
    },[cameraData, camera_qr_data])
```

4.4.12.3.6 Map Component

This component displays and interact with a dynamic map interface. included integrating a base64-encoded map image (*mapData*) to display real-time map data. Zoom functionality was implemented using buttons for zooming in (+) and out (-) (see code below), as well as mouse wheel support for smooth zooming. A loading state was added to display a placeholder message while the map data was being fetched. Using *useEffect*, the map image was dynamically updated whenever new *mapData* was received (see code below), ensuring real-time updates. CSS transformations (scale) were applied to enable smooth zooming transitions, and floating zoom buttons were designed for an intuitive user experience, styled with CSS for a modern look.

Code Block 4.4.9: Smooth zoom control for images

```
const zoomIn = () => {
      setZoom((prevZoom) => Math.min(prevZoom + 0.1, 7)); // Max zoom
       ↪  7x
  };

  const zoomOut = () => {
      setZoom((prevZoom) => Math.max(prevZoom - 0.1, 1)); // Min zoom
       ↪  1x (original size)
  };

  const handleWheel = (event) => {
      // Prevent the page from scrolling when zooming
```

```
        event.preventDefault();

        if (event.deltaY < 0) {
            zoomIn(); // Zoom in when scrolling up
        } else {
            zoomOut(); // Zoom out when scrolling down
        }
    };

    useEffect(() => {
        if (mapData) {
            const imageUrl = `${mapData}`;

            // Set image src only if imgRef.current is available
            imgRef.current.src = imageUrl;
            if(!mapData){
                setLoading(true);
            }
            setLoading(false)
            return () => {
                URL.revokeObjectURL(imageUrl);
            };
        }
    }, [mapData]);
```

4.4.12.3.7 Task Component

the Task component manages task creation and submission for the application. It is responsible for managing task creation and submission for the application. The responsibilities included designing and implementing a task creation interface that allowed users to select zones and define tasks, such as loading or unloading. The component dynamically built and displayed task sequences based on user input and integrated validation logic (see code below) to ensure task sequences adhered to predefined rules. For example, it prevented invalid task combinations like unloading without loading first and limited the number of tasks to a maximum of six. React hooks (*useState, useEffect, useCallback, useRef*) were utilized for state management and user interaction handling. Toast notifications (react-hot-toast) were integrated to provide real-time feedback for errors and successful actions. A mapping button was implemented to trigger mapping functionality and dynamically display available zones.

The component was styled using CSS and Tailwind CSS to achieve a clean and responsive design, ensuring a smooth and intuitive user experience while efficiently managing tasks within the application.

Code Block 4.4.10: Ensuring valid load/unload operations

```
const set = () => {
        if (params.length === 6) return
         ↪  toast.error("You can't load and unload more than 6 times");
        if (selected === "") return toast.error("Select a zone");
        let type = ref.current.value === "Loading" ? "L" : "U";
        if (type === "U") {
            let param = params[params.length - 1];
            if (param) {
                let key = Object.keys(param)[0];
                if (param[key] !== "Loading") {
                    return toast.error(
                     ↪  "You can't unload without loading");
                } else if (key === selected) {
                    return toast.error(
                     ↪  "You can't unload from a zone you've loaded from"
                     ↪  );
                }
            } else { return toast.error("You need to load first");}
        }else if( type === "L"){
            let param = params[params.length - 1];
            if (param) {
                let key = Object.keys(param)[0];
                if (param[key] === "Loading") {
                    return toast.error("You can't load twice");
                }
            }
        }
        setTask(`${task}${selected}(${type}) -> `);
        setParams([...params, { [selected]: ref.current.value }]);
        setSelected("");
    };
```

4.4.13 Robot Control Framework

The Robot Operating System (ROS) serves as the robot's main controller, managing all operations seamlessly at all times and facilitating communication with the GUI.

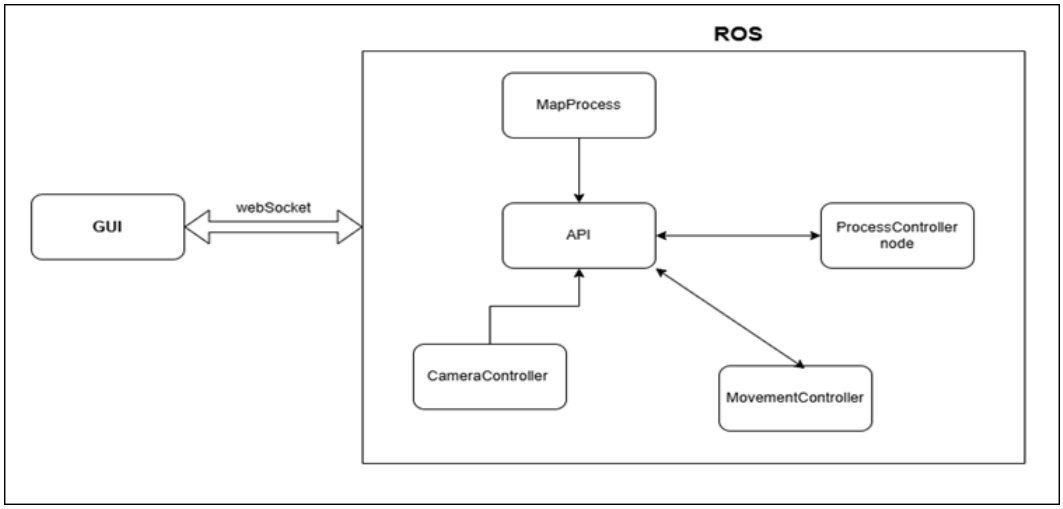


Figure 4.96: general architecture

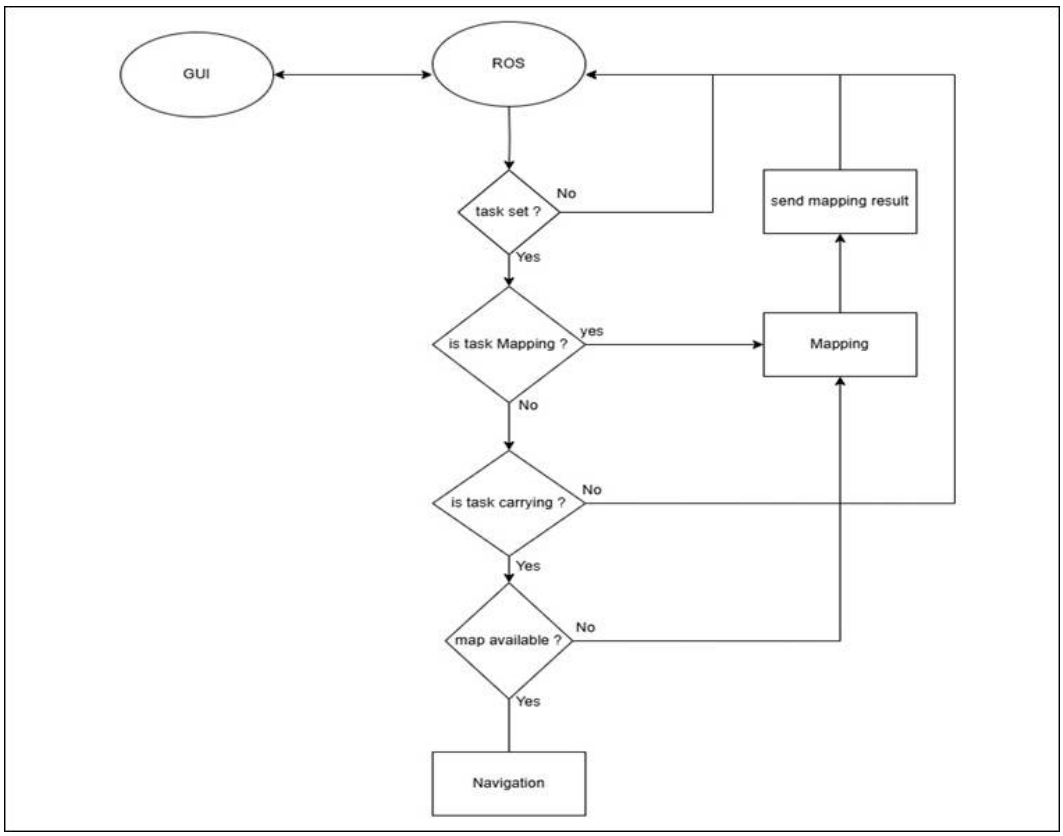


Figure 4.97: general architecture

4.4.13.1 Nodes

4.4.13.1.1 ie_API_Server

This node serves as a critical bridge between the ROS (Robot Operating System) ecosystem and a frontend application, leveraging *WebSocket* communication to enable seamless

interaction. It facilitates real-time data streaming, robot control, and task management, ensuring smooth and efficient communication between the backend and frontend (see fig. 4.96).

Code Block 4.4.11: API initial value

```
class ie_API_Server:
  def __init__(self):
      self._bridge = CvBridge()
      self.sio = socketio.async_server.AsyncServer(async_mode='asgi',
      ↪  cors_allowed_origins="*")
      self.app = socketio.asgi.ASGIApp(self.sio)
      self.mc_pub = rospy.Publisher("manual_controller", Int32,
      ↪  queue_size=10)
      self.cam_sub = rospy.Subscriber("camera_feed", Image,
      ↪  self.run_async_cameraFeedCallback)
      self.cam_qr_sub = rospy.Subscriber("camera_qr_code_feed",
      ↪  Image, self.run_async_cameraQrFeedCallback)
      self.sensors_sub = rospy.Subscriber("sensor_data",
      ↪  SensorDataMap, self.sensorsCallback)
      self.speed_sub = rospy.Subscriber("speed_value", Float32,
      ↪  self.run_async_speedCallback)
      self.map_sub = rospy.Subscriber("map_feed", Image,
      ↪  self.run_async_mapFeedCallback)
      self.process_sub = rospy.Subscriber("ProcessState", String,
      ↪  self.run_async_processState)
      rospy.Service('output', taskMessage,
      ↪  self.run_async_taskFinished)
      rospy.Service('mapping_output', mappingOutput,
      ↪  self.run_async_mappingOutput)
      rospy.Service('gear_changed', robotGear,
      ↪  self.run_async_gearChanged)
      self.sio.on("connect", self.onConnect)
      self.sio.on("disconnect", self.onDisconnect)
      self.sio.on("moveDirection", self.movement)
      self.sio.on("message", self.message)
      self.sio.on("robot_task", self.taskCallback)
      self.sio.on("robot_gear", self.gearCallback)
```

The node integrates *WebSocket communication* using *Socket.IO*, establishing a real-time, bidirectional channel that supports multiple clients. This allows users to connect, disconnect, and interact with the robot in real time. The WebSocket connection is the backbone of the system, enabling instant data exchange and control commands between the frontend and the

ROS backend.

Code Block 4.4.12: API initial value

```
  # Register event handlers
self.sio.on("connect", self.onConnect)
self.sio.on("disconnect", self.onDisconnect)
self.sio.on("moveDirection", self.movement)
self.sio.on("message", self.message)
self.sio.on("robot_task", self.taskCallback)
self.sio.on("robot_gear", self.gearCallback)
```

For *real-time data streaming*, the node subscribes to various ROS topics to capture and process critical information. It subscribes to *camera_feed* and *camera_qr_code_feed* to receive live camera images, which are then converted from ROS Image messages to OpenCV format using *CvBridge*. These images are encoded as *base64 strings* and streamed to the frontend via WebSocket. Similarly, the node subscribes to the *map_feed* topic to receive real-time map images, processes them, and streams them to the frontend in the same manner. Additionally, it subscribes to the *speed_value* topic to capture the robot's speed and emits speed updates to the frontend in real time, ensuring users have up-to-date information on the robot's movement.

The node also handles *robot control* by listening for specific events from the frontend. For movement control, it listens for *moveDirection* events (e.g., *UP, DOWN, LEFT, RIGHT, STOP*) and translates these into corresponding commands published to the *manual_controller* ROS topic, enabling direct control of the robot's movement. For gear control, it listens for *robot_gear* events to toggle between automatic and manual modes. This is achieved by calling the *change_gear* ROS service and emitting the updated gear state to the frontend, ensuring the user interface reflects the current state of the robot.

Task management is another key functionality of the node. It listens for *robot_task* events from the frontend, which trigger tasks such as loading, unloading, or mapping. The node converts task data into a ROS-compatible format (using the *TaskData* message) and calls the *robot_task* service to execute the task(see code block 4.4.12). It also listens for task feedback from the ROS backend via the *output* service, emitting task responses (e.g., success or failure messages) to the frontend. This ensures users receive timely updates on task progress and outcomes, enhancing transparency and usability.

For *mapping functionality*, the node listens for mapping data from the ROS backend via the *mapping_output* service. It processes and serializes this data, which includes information such as locations and coordinates, and emits it to the frontend. This allows users to visualize the robot's environment and track its movements in real time.

Finally, the node monitors the robot's *process state* by subscribing to the *ProcessState* topic. It captures updates on the robot's current state (e.g., Processing, Stationary) and emits these updates to the frontend in real time. This ensures users are always aware of the robot's operational status, enabling better decision-making and control.

Code Block 4.4.13: Handling async ROS tasks efficiently

```
  async def taskCallback(self, sid, task):
    t = TaskData()
    t.task_name = task['task']["task_name"]
    t.params = [
        Param(zone=key, type=value)
        for key, value in task['task']["params"].items()
    ]
    rospy.wait_for_service('robot_task')
    robot_task = rospy.ServiceProxy('robot_task', robotTask)
    robot_task.wait_for_service(10)
    try:
        response = robot_task(t)
        print(response)
        await self.sio.emit(
            "task_response",
            {"response": response.message},
            to=sid
        )
    except rospy.ServiceException as exc:
        print("Service did not process request: " + str(exc))

def run_async_taskFinished(self, output):
    try:
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        loop.run_until_complete(self.taskFinished(output))
    except Exception as e:
        rospy.logerr(f"
        ↪ Error before proccessing and emitting taskFinished: {e}")
    return taskMessageResponse(True)

async def taskFinished(self, output):
    await self.sio.emit("task_response", {"response": output.message})
```

```
def run_async_mappingOutput(self, output):
    try:
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        loop.run_until_complete(self.mappingOutput(output))
    except Exception as e:
        rospy.logerr(f"
        ↪ Error before proccessing and emitting taskFinished: {e}")
    return mappingOutputResponse(True)

async def mappingOutput(self, output):
    print(output)
    data = [
        {"location": kv.location, "x": kv.x, "y": kv.y}
        for kv in output.locations
    ]
    print(data)
    await self.sio.emit("mapping_output", {"locations": data})
```

4.4.13.1.2 movementController

This node manages the robot's movement in both *manual* and *autonomous* modes. It processes control commands, handles gear switching, and ensures smooth transitions between states.

Key Features and Functionality:

The node supports two modes of operation:

- *Manual Mode*: The robot is controlled via user commands, such as increasing or decreasing linear or angular velocity.
- *Autonomous Mode*: The robot is controlled by an external system, like a navigation stack, with manual control disabled.

To facilitate seamless switching between these modes, the *twist_mux* package can be utilized. This package subscribes to multiple *cmd_vel* topics.

The Movement Control component was designed to subscribe to the *manual_controller* topic to receive movement commands, such as *UP, DOWN, LEFT, RIGHT,* and *STOP*. The component adjusted the robot's linear and angular velocity based on the received commands and then published the resulting velocity commands to the *cmd_vel* topic.

Gear Switching system provides a ROS service (*change_gear*) to switch between manual and autonomous modes, updates the gear state and notifies the GUI via another ROS service (*gear_changed*) (see fig. 4.96). Velocity Interpolation Implements smooth interpolation for angular velocity to prevent abrupt changes in robot movement. Uses a waiting period and linear interpolation to gradually reduce angular velocity to zero.

Process State Monitoring Subscribes to the *ProcessState* topic to monitor the robot's current state (e.g., Processing, Stationary). Resets movement commands if the robot is in a Processing state. Speed Calculation Calculates the robot's total velocity by combining linear and angular velocity components. Publishes the calculated speed to the *speed_value* topic for real-time monitoring.

Technical Details:

- ROS Integration:
  - Publishers:
    - cmd_vel: Publishes velocity commands for the robot.
    - speed_value: Publishes the robot's current speed.
  - Subscribers:
    - manual_controller: Receives movement commands.
    - ProcessState: Monitors the robot's state.
    - cmd_vel: Receives velocity commands from the autonomous system.
  - Services:
    - change_gear: Handles gear switching between manual and autonomous modes.
- Data Structures:
  - cmd_vel Struct: Stores the robot's linear and angular velocity (see code block 4.4.12) components.Manages interpolation and waiting states for smooth velocity transitions.Provides methods to increase/decrease velocity and reset commands.
- Interpolation Logic: Uses timestamps (ros::Time) to track the last update time and interpolation duration. Implements a waiting period (wait_duration) before starting interpolation. Linearly interpolates angular velocity towards zero over a specified duration (interpolation_duration), see code block 4.4.11 for implementation.
- Velocity Calculation: Combines linear and angular velocity components to calculate the robot's total velocity. Accounts for the robot's wheel radius (radius) to convert angular velocity to linear velocity.

- Error Handling: Logs warnings for invalid commands or attempts to control the robot in autonomous mode. Ensures smooth transitions between states to prevent abrupt movements.

Code Block 4.4.14: Gear shift

```
bool changeGear (ie_communication::robotGear::Request &req,
 ↪  ie_communication::robotGear::Response &res){
  movedata.gear = req.state;
  if(movedata.gear == 0){
      std::cout << "
       ↪  The robot is in autonomous mode, manual control is disabled
       ↪  " << std::endl;
  }else{
      std::cout << "
       ↪  The robot is in manual mode, autonomous control is disabled
       ↪  " << std::endl;
  }

  ros::NodeHandle n;
  ros::ServiceClient client =
   ↪  n.serviceClient<ie_communication::robotGear>("gear_changed");
  ie_communication::robotGear srv;

  srv.request.state = movedata.gear;

  if(client.call(srv)){

  }else{
      ROS_ERROR("Failed to update the gear to GUI");
  }

  res.message = true;
  return true;
}
```

Code Block 4.4.15: Linear interpolation

```
void updateAngular() {
      ros::Time current_time = ros::Time::now();
      ros::Duration time_since_update = current_time -
       ↪  last_update_time;
```

```
        if (waiting_to_interpolate) {
            if (time_since_update.toSec() >= wait_duration) {
                // Start interpolation after waiting period
                angular_start = az;
                interpolation_start_time = current_time;
                angular_interpolating = true;
                waiting_to_interpolate = false;
            }
        }

        if (angular_interpolating) {
            ros::Duration time_since_start = current_time -
             ↪  interpolation_start_time;
            float t = time_since_start.toSec() /
             ↪  interpolation_duration;

            if (t >= 1.0) {
                az = 0.0; // Stop interpolation after duration
                angular_interpolating = false;
            } else {
                az = angular_start * (1.0 - t); // Linearly interpolate
                 ↪  towards zero
            }
        }
    }
```

Code Block 4.4.16: Cmd_vel structure

```
struct cmd_vel {
  int gear = 0;
  float lx = 0.0;
  float ly = 0.0;
  float lz = 0.0;
  float ax = 0.0;
  float ay = 0.0;
  float az = 0.0;

  ros::Time last_update_time; // Last time angular velocity was
   ↪  updated
```

```
ros::Time interpolation_start_time; // Time when interpolation
 ↪  starts
bool angular_interpolating = false; // Flag to indicate
 ↪  interpolation
bool waiting_to_interpolate = false; // Flag to indicate waiting
 ↪  period
float angular_start = 0.0; // Starting value for interpolation
float interpolation_duration = 0.8; // Duration for interpolation
 ↪  (in seconds)
float wait_duration = 0.3; // Time to wait before starting
 ↪  interpolation (in seconds)
float radius = 0.16;
void changeGear(const std_msgs::Int32::ConstPtr& msg) {
    gear = msg->data;
}
void increaseLinear() {
    lx += 0.01;
}

void decreaseLinear() {
    lx -= 0.01;
}

void increaseAngular() {
    az += 0.1;
    updateAngularState();
}

void decreaseAngular() {
    az -= 0.1;
    updateAngularState();
}

void reset() {
    lx = 0.0;
    ly = 0.0;
    lz = 0.0;

    ax = 0.0;
    ay = 0.0;
```

```
        az = 0.0;

        angular_interpolating = false;
        waiting_to_interpolate = false;
    }

    void updateAngularState() {
        last_update_time = ros::Time::now();
        angular_interpolating = false; // Stop interpolation if active
        waiting_to_interpolate = true; // Set waiting state
    }


    float getVelocity(){
        if (gear == 1){
            float linear_velocity_rotational = az * radius;
            float tVelocity = sqrt((pow(lx, 2) +
             ↪  pow(linear_velocity_rotational, 2)));
            return std::round(tVelocity * 100.0f) / 100.0f;
        }else{
            float linear_velocity_rotational = acvl.angular.z * radius;
            float tVelocity = sqrt((pow(acvl.linear.x, 2) +
             ↪  pow(linear_velocity_rotational, 2)));
            return std::round(tVelocity * 100.0f) / 100.0f;
        }
    }
};
```

4.4.13.1.3 CameraController

This node functions as a *camera feed manager*, playing a crucial role in subscribing to raw camera feeds, processing them, and publishing the processed feeds to other ROS topics for further use, such as display or analysis. It subscribes to two raw camera feeds: */camera/rgb/image_raw*, which is the primary camera feed used for general purposes like navigation or object detection, and */camera_2/camera_2/image_raw*, the secondary camera feed often utilized for specialized tasks such as QR code detection. The node then publishes the processed feeds to two ROS topics: *camera_feed* for the primary camera feed and *camera_qr_code_feed* for the secondary camera feed. This ensures that downstream nodes or applications have access to the necessary camera data for their respective tasks, see (code

block 4.4.16) for implementation.

The node also incorporates *camera state management* to ensure efficient operation. It uses a *boolean flag (_camState)* to enable or disable feed publishing, ensuring that feeds are only published when the camera is active and data is available. Additionally, it tracks camera availability *(_cam_available*) to prevent publishing when no data is being received. This state management mechanism helps optimize resource usage and prevents unnecessary processing. To handle potential issues, the node implements *robust error handling* for image conversion and publishing. Errors, such as failed image conversions or publishing failures, are logged using *rospy.logerr*, making it easier to debug and maintain the system.

From a technical perspective, the node integrates seamlessly with the ROS ecosystem. It subscribes to */camera/rgb/image_raw* and */camera_2/camera_2/image_raw* to receive raw images from the primary and secondary cameras, respectively. It then publishes the processed images to *camera_feed* and *camera_qr_code_feed* for further use. For image processing, the node uses *CvBridge* to convert ROS Image messages into OpenCV-compatible formats. While the node processes and publishes the images, it does not apply additional filtering or transformations, ensuring the images remain in their original state unless modified by downstream nodes.

To ensure smooth operation and responsiveness, the node is initialized in a *separate thread*. This threading approach allows the node to handle image processing and publishing without blocking the main thread, ensuring efficient performance even under high workloads. By combining ROS integration, state management, error handling, and threading, this camera feed manager node provides a reliable and efficient solution for managing and streaming real-time camera data within the ROS ecosystem. Its design ensures that downstream applications receive the necessary camera feeds promptly and accurately, making it an essential component for systems that rely on real-time visual data.

Code Block 4.4.17: CameraProcess node

```
import rospy
from ie_communication.srv import camState, camStateResponse
import cv2
from cv_bridge import CvBridge
from sensor_msgs.msg import Image

class CameraController:

    def __init__(self):
        self._pubcam1 = rospy.Publisher("camera_feed", Image,
        ↪  queue_size=10)
```

```
        self._pubcam2 = rospy.Publisher("camera_qr_code_feed", Image,
        ↪  queue_size=10)
        rospy.Subscriber("/camera/rgb/image_raw",Image,
        ↪  self._camera1Process)
        rospy.Subscriber("/camera_2/camera_2/image_raw",Image,
        ↪  self._camera2Process)
        self._bridge = CvBridge()
        self._camState = True
        self._cam_available = False
        self._cam1Frame = None
        self._cam2Frame = None

    def _camera1Process(self, data):
        try:
            self._cam1Frame = data
            self._cam_available = True
            self._provideCamFeed()
        except Exception as e:
            rospy.logerr(f"Error converting image: {e}")

    def _camera2Process(self, data):
        try:
            self._cam2Frame = data
            self._cam_available = True
            self._provideCam2Feed()
        except Exception as e:
            rospy.logerr(f"Error converting image: {e}")

    def _provideCamFeed(self) -> None:
        if self._camState and self._cam_available:
            try:
                self._pubcam1.publish(self._cam1Frame)
            except Exception as e:
                rospy.logerr(f"Error publishing image feed: {e}")

    def _provideCam2Feed(self) -> None:
        try:
            self._pubcam2.publish(self._cam2Frame)
        except Exception as e:
            rospy.logerr(f"Error publishing image from cam2 feed: {e}")
```

4.4.13.1.4 map_process

This node is responsible for processing laser scan data, robot position, and QR code positions to generate and publish a dynamic map for visualization and navigation. It subscribes to the */scan* topic to receive laser scan data, which is processed to detect obstacles and update the local map. The robot's position and orientation are tracked using *TF2*, which retrieves the robot's translation and rotation in the map frame. These coordinates are then converted into pixel coordinates for visualization on the map. Additionally, the node integrates QR code positions by reading them from an SQLite database (*qr_code.db*) every 5 seconds. These QR code positions are displayed on the map as blue rectangles, providing a comprehensive view of the environment.

Code Block 4.4.18: Update Map from Scan Function

```
def update_map_from_scan(self):
  if self.robot_position is None or self.robot_rotation is None:
     return
  yaw = self.get_yaw_from_quaternion(self.robot_rotation)

  for i, range_val in enumerate(self.latest_scan.ranges):
      if range_val < self.latest_scan.range_min or range_val >
      ↪  self.latest_scan.range_max:
         continue  # Skip invalid range values

angle=self.latest_scan.angle_min+i* self.latest_scan.angle_increment

      endpoint_x = self.robot_position.x + range_val * np.cos(yaw +
      ↪  angle)
      endpoint_y = self.robot_position.y + range_val * np.sin(yaw +
      ↪  angle)

      endpoint_x_pixel = int((endpoint_x - self.map_origin_x) /
      ↪  self.map_resolution)

      endpoint_y_pixel = int((endpoint_y - self.map_origin_y) /
      ↪  self.map_resolution)

      if 0 <= endpoint_x_pixel < self.map_width and 0 <=
      ↪  endpoint_y_pixel < self.map_height:

            self.map_image[endpoint_y_pixel, endpoint_x_pixel] = 0
```

The node generates a *2D map image* that includes several key elements: obstacles derived from laser scan data (represented as black pixels), the robot's current position (shown as a red circle) and orientation (indicated by a green line), QR code positions (displayed as blue rectangles), and the robot's path (traced as a green line over time). This map image is published to the *map_feed* topic, enabling real-time visualization for users or other nodes. The map is dynamically updated at a fixed rate of 10 Hz using a timer callback, ensuring it reflects the latest scan data, robot position, and QR code positions. The map parameters, such as resolution (5 cm per pixel), dimensions (*400x400 pixels*), and origin *(-10.0, -10.0 meters*), are predefined, and the map is initialized as free space (white pixels) before being updated with obstacles (black pixels) from laser scans.

In summary, this node provides a dynamic and real-time map visualization by integrating laser scan data, robot position, and QR code positions. Its robust processing and integration capabilities ensure accurate and up-to-date map generation, making it a vital component for navigation and visualization tasks in the ROS ecosystem.

Code Block 4.4.19: Function Drawing Map

```
def process_scan(self):
      self.map_image.fill(255)

      robot_x_pixel = int((self.robot_position.x -
       ↪  self.map_origin_x) / self.map_resolution)
      robot_y_pixel = int((self.robot_position.y -
       ↪  self.map_origin_y) / self.map_resolution)

      if not (0 <= robot_x_pixel < self.map_width and 0 <=
       ↪  robot_y_pixel < self.map_height):
          return
    self.update_map_from_scan()
    map_image_color=cv2.cvtColor(self.map_image, cv2.COLOR_GRAY2BGR)

    if len(self.path_history) > 1:
        for i in range(1, len(self.path_history)):
              cv2.line(map_image_color, self.path_history[i - 1],
               ↪  self.path_history[i], (0, 255, 0), 2)

    for qr_x, qr_y in self.qr_code_positions:
```

```
            qr_x_pixel=int((qr_x-self.map_origin_x)/ self.map_resolution)
            qr_y_pixel=int((qr_y-self.map_origin_y)/ self.map_resolution)
              if 0 <= qr_x_pixel < self.map_width and 0 <= qr_y_pixel <
               ↪  self.map_height:
                  cv2.rectangle(map_image_color,
(qr_x_pixel - self.qr_code_size, qr_y_pixel -
 ↪  self.qr_code_size),(qr_x_pixel + self.qr_code_size, qr_y_pixel +
 ↪  self.qr_code_size),(255, 0, 0), -1)

          cv2.circle(map_image_color, (robot_x_pixel, robot_y_pixel), 5,
           ↪  (0, 0, 255), -1)
          yaw = self.get_yaw_from_quaternion(self.robot_rotation)
          robot_x_end = robot_x_pixel + int(np.cos(yaw) * 10)
          robot_y_end = robot_y_pixel + int(np.sin(yaw) * 10)
          cv2.line(map_image_color, (robot_x_pixel, robot_y_pixel),
           ↪  (robot_x_end, robot_y_end), (0, 255, 0), 2)

          map_image_color = np.flipud(map_image_color)
          map_image_color = cv2.rotate(map_image_color,
           ↪  cv2.ROTATE_90_COUNTERCLOCKWISE)
          output_path = "/tmp/robot_map_with_position.png"
          cv2.imwrite(output_path, map_image_color)
          self.send_image(output_path)
```

4.4.13.2 Classes

4.4.13.2.1 Task

The Task class is a base class that encapsulates common functionality for robot tasks, such as: Robot *control* (movement, turning, stopping), *Obstacle detection and avoidance* using LIDAR data, *Image processing* for line following and QR code detection, *State management* for task execution and transitions.

Key Features and Functionality:

The robot control system offers a comprehensive set of methods for managing basic robot movements. These include *_move_forward*, which propels the robot forward at a constant speed, as well *as _turn_left* and *_turn_right* for executing left and right turns, respectively. Additionally, the system features a *_U_turn* method for performing a 180° turn and a *_move* function that adjusts the robot's movement based on error, particularly useful for tasks like line following. To halt the robot, the *stop* method is available.

For obstacle detection and avoidance, the system subscribes to the */scan* topic to receive

LIDAR data, enabling it to monitor the surroundings. It employs the *check_for_obstacles* function (see code block 4.4.17) to detect obstacles in the robot's path. When an obstacle is detected, the system utilizes a contouring algorithm called *contour_obstacle(see* code block 4.4.17*)* to navigate around it. This algorithm involves turning 90° left, moving forward, turning 90° right, and then returning to the original path. To ensure precise turns, the system tracks the robot's orientation using odometry data, accessed through the *_get_yaw* method. This combination of movement control and obstacle avoidance ensures efficient and accurate navigation in dynamic environments.

Code Block 4.4.20: Obstacle avoidance procedure

```
def contour_obstacle(self):
        self._contourningStep += 1
        if self.timer:
            self.timer.shutdown()
        self.stop()  # Stop the robot

        if self._contourningStep == 1:
            rospy.loginfo("Contouring obstacle: Step 1")
            self.turn_angle(90)
            self.contour_obstacle()
        elif self._contourningStep == 2:
            rospy.loginfo("Contouring obstacle: Step 2")
            self._move_forward()
            while not self.is_obstacle_behind():
                rospy.sleep(0.1)
            self.turn_angle(-90)  # Turn right 90°
            self.contour_obstacle()
        elif self._contourningStep == 3:
            rospy.loginfo("Contouring obstacle: Step 3")
            self._move_forward()
            while not self.is_obstacle_behind():
                rospy.sleep(0.1)
            self.turn_angle(-90)
            self.contour_obstacle()
        elif self._contourningStep == 4:
            rospy.loginfo("Contouring obstacle: Step 4")
            self._move_forward()
            self._turnSide = "left"
            self.moveToTurnPosition = True
            self._obstacleInFront = False
```

Code Block 4.4.21: Function checking if the robot overpass the obstacle

```
    def is_obstacle_behind(self):
            if self.scan_data is None:
                return False
            robot_length = 0.52
            obstacle_distance_threshold = robot_
            sector_start_angle = np.deg2rad(225)  # 225 degrees
            sector_end_angle = np.deg2rad(270)   # 270 degrees
            angle_increment = self.scan_data.angle_increment
            num_ranges = len(self.scan_data.ranges)
            start_index = int((sector_start_angle -
             ↪  self.scan_data.angle_min) / angle_increment)
            end_index = int((sector_end_angle - self.scan_data.angle_min) /
             ↪  angle_increment)
            min_distance = float('inf')
            for i in range(start_index, end_index):
                if 0 < self.scan_data.ranges[i] < min_distance:
                    min_distance = self.scan_data.ranges[i]
            print(f"min distance : {min_distance}")
            return min_distance > obstacle_distance_threshold and
             ↪  min_distance < (obstacle_distance_threshold + 0.5)
```

Code Block 4.4.22: Obstacle Detection

```
    def check_for_obstacles(self, event):
      if self.scan_data is None or len(self.scan_data.ranges) == 0:
        return

  print("Checking for obstacles using LIDAR data")
    front_angle_range = 30  # Degrees
      front_angle_range_rad = np.deg2rad(front_angle_range)
      angle_increment = self.scan_data.angle_increment
      num_ranges = len(self.scan_data.ranges)
      min_angle = self.scan_data.angle_min
      start_index = max(0, int((min_angle - front_angle_range_rad / 2) /
       ↪  angle_increment))
      end_index = min(num_ranges - 1, int((min_angle +
       ↪  front_angle_range_rad / 2) / angle_increment))
      min_distance = float('inf')
      for i in range(start_index, end_index):
```

```
        if 0 < self.scan_data.ranges[i] < min_distance and
        ↪  np.isfinite(self.scan_data.ranges[i]):
            min_distance = self.scan_data.ranges[i]
            obstacle_distance_threshold = 0.3
    if min_distance < obstacle_distance_threshold:
        self._obstacleInFront = True
        rospy.loginfo(f"Obstacle detected at {min_distance:.2f}
        ↪   meters!")
        self.stop()  # Stop the robot
        self._obstacleChecker += 1

    if self._obstacleChecker >= 5:
      rospy.loginfo("
      ↪  Obstacle is still present. Calling contour_obstacle function.
      ↪  ")
      self._obstacleChecker = 0

    if not self._contourningObstacle:
        self._contourningObstacle = True
        self.contour_obstacle()
    else:
        self._obstacleInFront = False
        self._obstacleChecker = 0
        rospy.loginfo("No obstacle detected. Moving.")
```

The image processing component of the system is designed to enhance the robot's navigation and interaction capabilities by subscribing to multiple camera topics, such as */camera/rgb/image_raw* and */camera_qr_code_feed*. This allows the robot to perform tasks like line following and QR code detection. For line following, the system detects black lines using techniques such as *thresholding* and *contour detection*, enabling the robot to accurately track and follow predefined paths. Additionally, the system incorporates QR code detection using the *QReader library*, which decodes QR codes to extract relevant information or commands. To ensure precise alignment with detected lines, the system implements the *_adjust_orientation* method, which dynamically adjusts the robot's movement based on real-time line detection data. Together, these features enable the robot to navigate complex environments and interact with visual markers effectively.

The task state management system is responsible for monitoring and controlling the execution of tasks. It tracks the task's current state using the *_running* variable, which indicates whether the task is active or inactive. The system provides two key methods: *start* to initiate

the task and *stop* to halt it, ensuring flexibility and control over task execution. Additionally, it emits signals such as *finishedSignal* and *failedSignal* to notify the system when a task has been successfully completed or has encountered a failure. These signals enable the system to respond appropriately, whether by transitioning to the next task or handling errors, ensuring smooth and efficient operation.

The system incorporates *robust error handling* to manage potential issues across various components, including ROS callbacks, image processing, and task execution. Errors and task statuses are logged using *rospy.loginfo* for informational messages and *rospy.logerr* for error reporting, ensuring transparency and ease of debugging.

To enhance efficiency and responsiveness, the system leverages *asynchronous execution* through the *asyncio* library. The *_execute* method is designed to run tasks in a non-blocking manner, allowing the robot to perform multiple operations simultaneously without interruptions. This approach ensures smooth operation and maintains system responsiveness, even during complex or time-consuming tasks.

Technical Details:

The system is tightly integrated with ROS (Robot Operating System) to facilitate seamless communication and control. It utilizes *subscribers* to receive critical data from various sensors and topics:

- */odom*: Provides odometry data, enabling the system to track the robot's position and orientation.
- */scan*: Delivers LIDAR data, which is essential for obstacle detection and avoidance.
- */camera/rgb/image_raw, /camera_qr_code_feed*, and other camera topics: Supply image feeds for tasks like line following and QR code detection.

On the output side, the system employs *publishers* to send commands and data:

- */cmd_vel*: Publishes velocity commands to control the robot's movement, such as forward motion, turns, and stops.
- */error*: Publishes error values for debugging and monitoring system performance.
- */position_joint_controller/command*: Sends commands to control lifting mechanisms or other actuators, if applicable.

This integration ensures efficient data flow and real-time control, enabling the robot to perform complex tasks with precision and reliability.

4.4.13.2.1.1 Navigation system

The navigation system is composed of three key functions: *_adjust_orientation*, *check_side_pixels*, and *check_straight_pixels*. These methods work together to enable the robot to follow lines and detect junctions effectively, ensuring smooth and accurate navigation in its environment.

The *_adjust_orientation* method is responsible for adjusting the robot's orientation based on the detected line and the number of black pixels in the camera image. It ensures the robot stays centered on the line or makes appropriate turns at junctions. This method takes several input parameters, including the error (deviation of the line from the center of the image), the angle of the detected line, the number of black pixels in the region of interest, the binary mask of the image, and the dimensions of the bounding box around the detected line. If the robot is moving to a lift position, it adjusts its orientation to reach the target. Otherwise, it checks the number of black pixels to determine the robot's position relative to the line. If the number of black pixels is within a defined threshold, the robot moves forward while adjusting its orientation based on the error. If the number of black pixels exceeds the threshold, the robot uses *check_side_pixels* and *check_straight_pixels* to determine if it's at a junction. Based on the detected line configuration, the robot decides whether to turn left, right, or move forward. The method then publishes velocity commands to the */cmd_vel* topic to adjust the robot's movement.

Code Block 4.4.23: Adjust Orientation Function

```
def _adjust_orientation(self, error, angle, pixels, mask , w_min,
↪ h_min):
    if self._obstacleInFront:
        return None
    if self.moveToTurnPosition:
        self._move(error)
        return
    if self._making_turn:
        print("error", error)
        if error < 60 or error > -60:
            self.stop()
            self._making_turn = False
        return
    if self._making_u_turn:
        print("error", error)
        if error < 60 or error > -60:
            self.stop()
            self._making_u_turn = False
```

```
            return
        if (w_min < 100 and h_min < 100):
            self._move_forward()
            return None
        if (pixels <= self._black_pixels + (self._black_pixels *
        ↪  self._threshold)) and  (pixels >= self._black_pixels -
        ↪  (self._black_pixels * self._threshold)):
            self._move(error)
            return None
        if pixels > (self._black_pixels + (self._black_pixels *
        ↪  self._threshold)):
            left_pixels, right_pixels, onLeft, onRight =
            ↪  self.check_side_pixels(mask)
            if onLeft or onRight:
                top_pixels_remaining, top_pixels_side,
                ↪  bottom_pixels_side, onTop, hastoStop =
                ↪  self.check_straight_pixels(mask)
                if hastoStop:
                    self.needMakeDecision = True
                    self.junction_decision(onLeft, onRight, onTop)
                    if top_pixels_remaining == 0:
                        self._move_forward()
                    return [left_pixels, right_pixels,
                    ↪  top_pixels_remaining, top_pixels_side,
                    ↪  bottom_pixels_side]
                self._move_forward()
                return [left_pixels, right_pixels, 0, 0, 0]
            elif pixels < (self._black_pixels - (self._black_pixels *
            ↪  self._threshold)):
                self._move(error)
                return None
        self._move(error)
        return None
```

The *check_side_pixels* method checks the number of black pixels on the left and right sides of the image to detect junctions or turns. It takes the binary mask of the image as input and divides the image into left and right halves, excluding the middle region. It counts the number of black pixels in each half and determines if the number of black pixels on either side exceeds a threshold, indicating a potential turn. The method returns the number of black pixels on the left and right sides, along with flags indicating whether a turn is detected. This

information is used *by _adjust_orientation* to make decisions at junctions.

Code Block 4.4.24: Check Side Pixels Function

```
def check_side_pixels(self, mask):
    onLeft = False
    onRight = False
    height, width = mask.shape

    # Defining middle region of the mask
    middle_start = (width // 2) - (self._middle_width // 2)
    middle_end = (width // 2) + (self._middle_width // 2)

    # Create a mask for the middle region and remove it from the
    ↪ original mask
    middle_mask = np.zeros_like(mask)
    middle_mask[:, middle_start:middle_end] = 255
    masked_binary = cv2.bitwise_and(mask, cv2.bitwise_not(middle_mask))

    # Split the masked binary into left and right halves
    left_half = masked_binary[:, :width // 2]
    right_half = masked_binary[:, width // 2:]

    # Count the black pixels in both halves
    left_pixels = np.sum(left_half == 255)
    right_pixels = np.sum(right_half == 255)

    # Define a threshold for the number of black pixels to consider
    ↪ left or right
    threshold_lr = (self._black_pixels // 3) - 20

    # Check if the left or right region has sufficient black pixels
    if left_pixels > threshold_lr:
        onLeft = True
    if right_pixels > threshold_lr:
        onRight = True

    return left_pixels, right_pixels, onLeft, onRight
```

The *check_straight_pixels* method checks the number of black pixels in the top and bottom halves of the image to detect straight paths or junctions. It also takes the binary mask of

the image as input and divides the image into top and bottom halves, excluding the middle region. It counts the number of black pixels in each half and determines if the number of black pixels in the bottom half exceeds the top half, indicating a potential junction. If a junction is detected, it checks the continuity of the line in the top half to determine if the robot should move forward. The method returns the number of black pixels in the top and bottom halves, along with flags indicating whether the robot should stop or move forward.

Code Block 4.4.25: Check Straight Pixels Function

```
def check_straight_pixels(self, mask):
    onTop = False
    hastoStop = False
    height, width = mask.shape
    middle_start = (width // 2) - (self._middle_width // 2)
    middle_end = (width // 2) + (self._middle_width // 2)

    # Create a mask for the middle region and remove it from the
    ↪ original mask
    middle_mask = np.zeros_like(mask)
    middle_mask[:, middle_start:middle_end] = 255
    masked_binary = cv2.bitwise_and(mask, cv2.bitwise_not(middle_mask))

    # Split the masked binary into top and bottom halves
    top_half = masked_binary[:height // 2, :]
    bottom_half = masked_binary[height // 2:, :]

    # Count the white pixels in the top and bottom halves
    top_pixels = np.sum(top_half == 255)
    bottom_pixels = np.sum(bottom_half == 255)

    # Check if bottom pixels are greater than or equal to top pixels,
    ↪ implying a need to stop
    if bottom_pixels >= top_pixels:
        hastoStop = True
        middle_line = mask[:, middle_start:middle_end]
        height, _ = middle_line.shape

        # Split the middle line into top and bottom portions
        top_half = middle_line[:height // 2, :]
        bottom_half = middle_line[height // 2:, :]
```

```
        # Zero out the black pixels in the bottom portion
        bottom_half[bottom_half == 255] = 0

        # Count remaining white pixels in the top half
        top_pixels_remaining = np.sum(top_half == 255)
        continuity_threshold = 0.3 * self._black_pixels

        # Check if top pixels remaining are above the threshold
        if top_pixels_remaining >= continuity_threshold:
            onTop = True

    return top_pixels_remaining, top_pixels, bottom_pixels, onTop,
    ↪ hastoStop
```

Together, these methods form a robust navigation system that allows the robot to follow lines accurately and make informed decisions at junctions. The *_adjust_orientation* method continuously adjusts the robot's movement based on real-time data, while *check_side_pixels* and *check_straight_pixels* provide critical information about the robot's surroundings. This combination ensures the robot can navigate complex environments with precision and reliability, adapting its behavior based on the detected line and junction configurations.

4.4.13.2.1.2 Interaction Between Methods:

Line Following: The *_adjust_orientation* method uses *check_side_pixels* and *check_straight_pixels* to determine if the robot is at a junction or should continue following the line. If the robot is on a straight path, it adjusts its orientation based on the error and moves forward.

Junction Detection: When the number of black pixels exceeds the threshold, *check_side_pixels* and *check_straight_pixels* are called to detect junctions. Based on the results, the robot makes a decision to turn left, right, or move forward.

Navigation: *The junction_decision* method (in child classes like *Mapping* and *Carrying*) uses the output of *check_side_pixels* and *check_straight_pixels* to navigate junctions and reach goal zones.

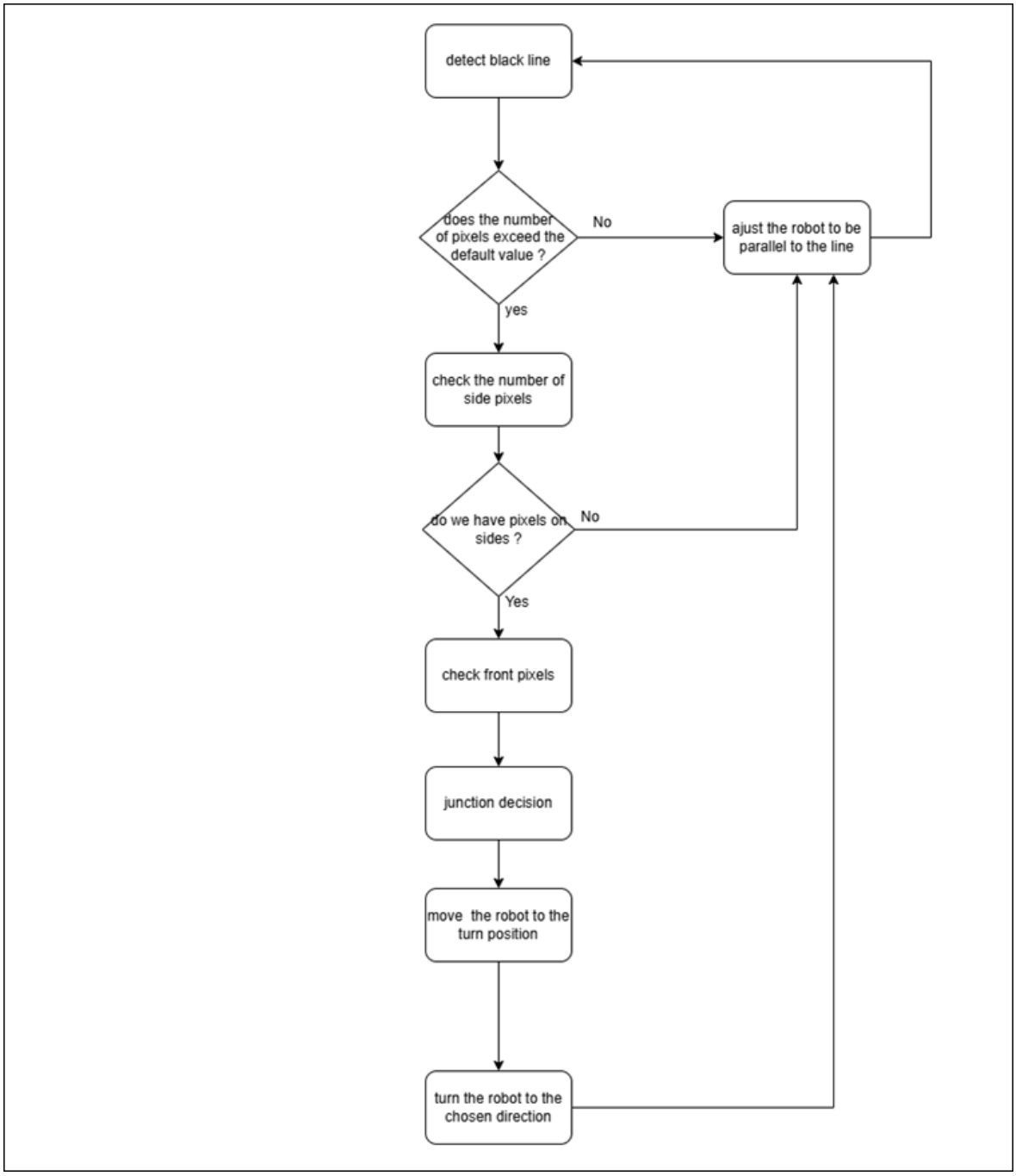


Figure 4.98: line following & junction flowchart

4.4.13.2.1.3 Example Scenario

- *Following a Straight Line: The robot detects a line with a small error and a number of black pixels within the threshold. The _adjust_orientation method adjusts the robot's orientation and moves it forward (see* code block 4.4.25 Robot following the line and detecting a junction*)* .
- *Approaching a Junction: The robot detects a significant increase in black pixels, indicating a junction.* The *check_side_pixels* method detects more black pixels on the left side, suggesting a left turn. The *check_straight_pixels* method confirms that the robot should stop and make a decision. The *junction_decision* method instructs the robot to turn left, so the robot move to the turn positrion (see fig. 4.98 the robot is at the turn position), and turn to the choosen direction (see fig. 4.99 The robot turns to the chosen direction).
- Navigating a Complex Path: The robot uses a combination of *check_side_pixels, check_straight_pixels,* and *junction_decision* to navigate through multiple junctions and reach its goal.

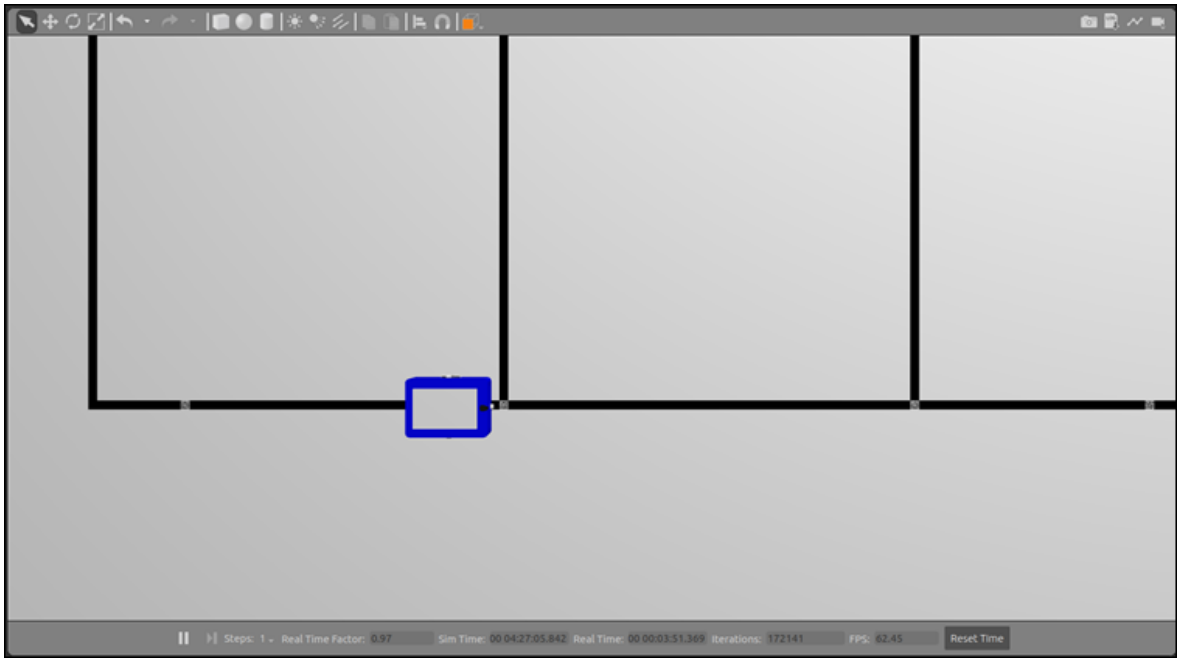


Figure 4.99: Robot following the line and detecting a junction

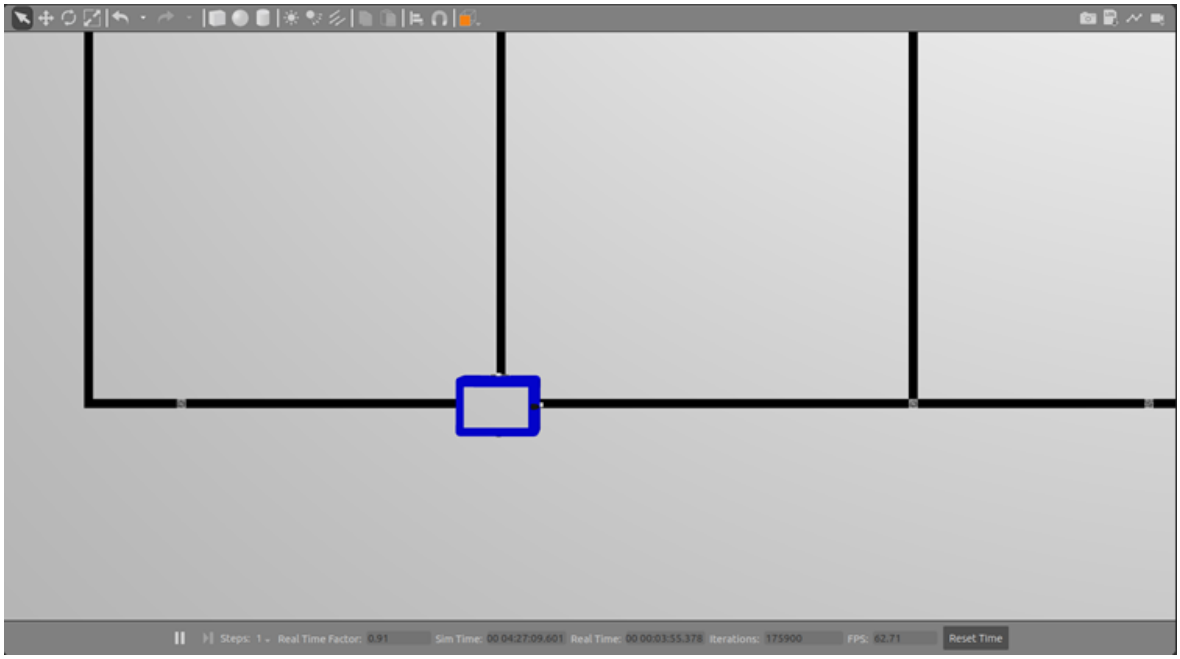


Figure 4.100: the robot is at the turn position

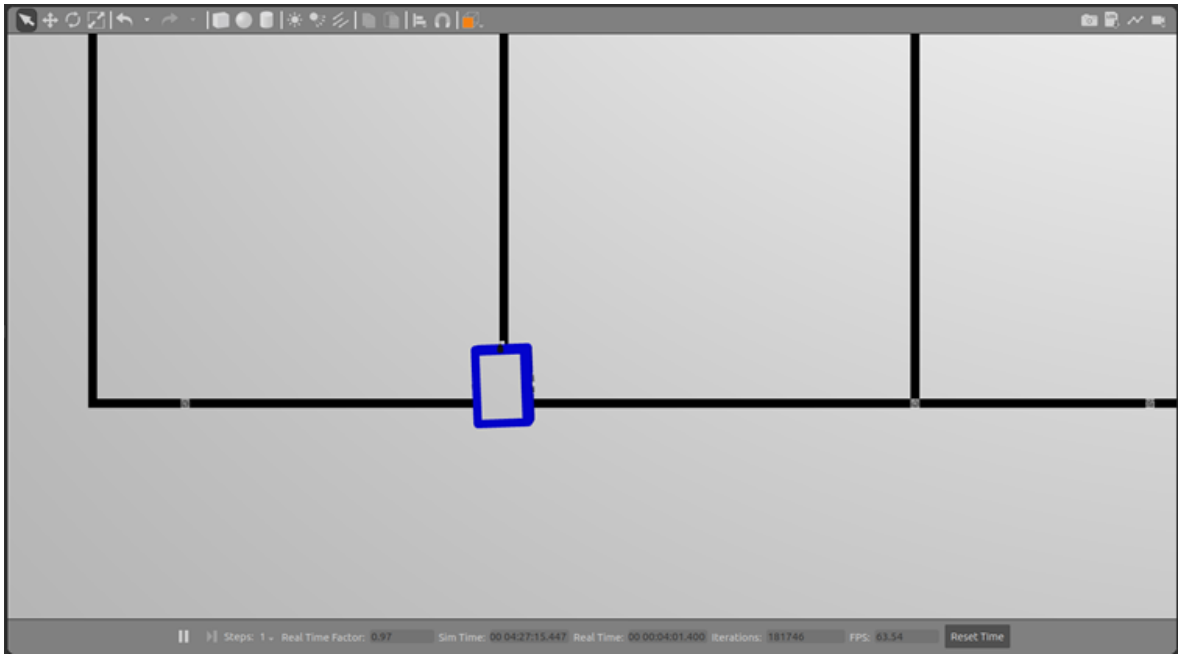


Figure 4.101: The robot turns to the chosen direction

4.4.13.2.1.4 Essentials parameters

The parameters: *self.param self._black_pixels self._sideBlackPixels self._threshold self._middle_width self.distanceToQr* and *obstacle_distance_threshold* are critical to the functionality of the Task class. These values were carefully tuned through *extensive testing and experimentation* to ensure optimal performance. Let's break down each parameter, its purpose, and how it was determined:

_black_pixels: Represents the expected number of black pixels in the camera image when the robot is following a straight the line. With a default Value of 493850, this value was determined by analyzing the camera image when the robot is perfectly aligned on the line. This parameter is used as a reference value for line-following logic. If the number of black pixels deviates significantly from this value, the robot adjusts its orientation.

_sideBlackPixels: Represents the expected number of black pixels on the sides of the camera image when the robot is approaching a turn. Its Value is 414556, Determined by analysing the camera image when the robot is approaching a junction or turn.

It's used to detect when the robot should initiate a turn (e.g., when the number of black pixels on one side exceeds this threshold).

_threshold: Defines the acceptable deviation from self._black_pixels for line-following.

The correct value found after test is 0.119 (11.9%). This parameter determined through experimentation to balance sensitivity and robustness. If the number of black pixels deviates by more than 11.9% from self._black_pixels, the robot adjusts its orientation.

_middle_width: Defines the width of the region of interest (ROI) in the camera image for line detection. Its value is 580 (pixels width). Set to focus on the central portion of the camera image, where the line is most likely to be.Adjusted to ensure the robot can detect the line even if it deviates slightly from the center.

distanceToQr: Represents the distance (in meters) from the robot to the QR code when it is detected. With a value of 0.3869 meters, determined by measuring the distance at which the QR code is reliably detected and decoded and ensures the robot stops at an appropriate distance from the QR code for accurate processing.

obstacle_distance_threshold: Defines the minimum distance (in meters) at which an obstacle is considered too close and requires avoidance. Its value is 0.3 (meters).

Determined through testing to balance safety and efficiency. A smaller value could risk collisions, while a larger value could cause unnecessary detours.

4.4.13.2.1.5 How These Values Were Determined:

- *Iterative Testing*: Each parameter was initially set to a rough estimate based on theoretical calculations or prior experience. The robot was then tested in various scenarios

(e.g., line following, obstacle avoidance, QR code detection) to observe its behaviour.

- *Incremental Adjustments*: Parameters were adjusted incrementally to improve performance. For example: Adjusting *self._threshold* to reduce oscillations during line following.
- *Real-World Validation*: The robot was tested in real-world environments (e.g., with varying lighting conditions, uneven surfaces, and different obstacle configurations) to ensure robustness.
- Trade-Offs: Some parameters required trade-offs. For example: A larger *obstacle_distance_threshold* increases safety but may result in longer detours.

4.4.13.2.1.6 Child Classes: Mapping and Carrying

The child classes (Mapping and Carrying) will extend the Task class to implement task-specific logic.

Mapping: Focuses on exploring the environment, detecting QR codes, and building a map.

Carrying: Focuses on transporting items between specified zones.

4.4.13.2.2 Mapping

The *Mapping class* is designed to explore the environment, detect QR codes, and store their positions in a database. It inherits from the *Task class*, leveraging its core functionality such as robot control, obstacle avoidance, and line following, while adding mapping-specific logic to fulfill its unique role. This inheritance allows the Mapping class to reuse existing methods and focus on extending functionality for environment exploration and QR code detection.

One of the key features of the Mapping class is *QR code detection*, which is handled by the *_check_qr* method. This method detects QR codes using image processing techniques and stores the detected QR codes along with their positions in a dictionary (*self.qrcodes*). To prevent duplicate detections, the class compares new QR codes with the last detected one (*self._lastQrCode*), ensuring that each QR code is only recorded once. This avoids redundancy and improves the efficiency of the mapping process.

For *environment exploration*, the class utilizes the *junction_decision* method to navigate junctions systematically. When a junction is detected (e.g., left, right, or top), the robot makes a decision to turn or move forward based on the exploration strategy. This ensures that the robot explores all possible paths in the environment, leaving no area unmapped.

The systematic approach guarantees comprehensive coverage, which is essential for accurate mapping.

The Mapping class also integrates with an *SQLite database* (qr_code.db) to store detected QR codes and their positions. The *_store* method is responsible for inserting QR code data into the database, while the *_checkExistingQrCodes* method checks for existing QR codes to avoid redundant entries. This ensures that the database remains up-to-date and free of duplicates. The database schema includes a table named *qr_code* with columns for id (primary key), name (QR code identifier), position_x (X-coordinate), and position_y (Y-coordinate), providing a structured way to store and retrieve mapping data.

When the mapping process is complete, the class emits a signal (*fsignal*) containing the detected QR codes, notifying other system components of the task's completion. It also calls the parent class's *_task_finished* method to clean up resources and notify the system, ensuring a smooth transition to the next task. The use of the *signalslot library* enables decoupled communication between the Mapping class and other components, enhancing modularity and flexibility.

The class implements *robust error handling* to manage potential issues during database operations, such as table creation or data insertion. Errors are logged using *rospy.logerr*, providing detailed information for debugging and maintenance. Additionally, the class ensures that the database connection is properly closed, preventing resource leaks and ensuring data integrity.

4.4.13.2.2.1 How It Works

The *Mapping class* operates through a well-defined workflow that ensures systematic environment exploration, QR code detection, and data storage. Here's how it works in detail:

Initialization

The Mapping class begins by initializing itself through the parent class's constructor using super().__init__("mapping"). This sets up the core functionality inherited from the *Task class*, such as robot control, obstacle avoidance, and line following. Additionally, the class initializes the *fsignal*, a signal used to notify other components when the mapping task is complete. This signal is essential for decoupled communication within the system.

Start Mapping

The mapping process is initiated by calling the *start* method. This method first checks for existing QR codes in the SQLite database using the *_checkExistingQrCodes* method. If QR codes are already present in the database, the task completes immediately by emitting the *fsignal* with the existing data. This avoids redundant mapping and saves time. If no QR codes are found, the robot begins exploring the environment, systematically navigating through the space to detect and record new QR codes.

QR Code Detection

QR code detection is handled by the *_check_qr* method. When a QR code is detected, the method processes it by converting its position to global coordinates using the *_calculate_distance* function. This ensures that the QR code's location is accurately recorded relative to the robot's environment. The detected QR code and its position are then stored in the *self.qrcodes* dictionary. To prevent duplicate detections, the method compares the new QR code with the last detected one (*self._lastQrCode*). If the QR code has already been recorded, it is ignored, ensuring that only unique QR codes are stored.

Code Block 4.4.26: Mapping Qr Code Checker

```
def _check_qr(self, decoded_text):
  if not self._processQrCode :
    self._processQrCode = True
    if decoded_text[0] == "None" or decoded_text[0] is None:
        self._processQrCode = False
        return
    if decoded_text[0] != self._lastQrCode:
        if len(self.qrcodes) >= 1:
          if (decoded_text[0] == list(self.qrcodes.keys())[0]):
              self._processQrCode = False
              self._task_finished(message="Mapping process finished")
              return
        self.hasDetectedQrRecently = False
        position = self._calculate_distance(self.robot_pose)
        self.qrcodes[decoded_text[0]] = position
        self._lastQrCode = decoded_text[0]
        self._processQrCode = False
    self._processQrCode = False
```

Junction Navigation

The *junction_decision* method plays a critical role in navigating the robot through the environment. When the robot encounters a junction, this method determines the appropriate action based on the type of junction detected:

- If a *top junction* is detected, the robot moves forward to continue exploration.
- If a *left or right junction* is detected, the robot prepares to turn, ensuring that all paths are systematically explored. This method ensures comprehensive coverage of the environment, leaving no area unmapped.

Code Block 4.4.27: Junction decision of mapping class

```
def junction_decision(self, onLeft, onRight, onTop):
    tm = 3
    if onTop:
    print("On top")
    tm = 1
    self._move_forward()
    self.timer= rospy.Timer(rospy.Duration(tm),self.resume_processing)
    elif onLeft:
        self._turnSide = "left"
        self.moveToTurnPosition = True
    elif onRight:
        self._turnSide = "right"
        self.moveToTurnPosition = True
```

Task Completion

Once the mapping process is complete, the *_task_finished* method is called to wrap up the task. This method performs several key actions:

1. It stores the detected QR codes in the SQLite database using the *_store* method, ensuring that the data is saved for future use.

2. It emits the *fsignal* with the QR code data, notifying other system components that the task is complete.

3. It calls the parent class's *_task_finished* method to clean up resources and notify the system, ensuring a smooth transition to the next task.

4.4.13.2.2.2 Impact on the System

The Mapping class enables the robot to *autonomously explore its environment*, *detect QR codes*, and *store their positions* for future use. It leverages the parent class's functionality for robot control and obstacle avoidance, demonstrating the power of *inheritance* and *code reuse*. The integration with SQLite ensures *persistent storage* of mapping data, enabling other tasks (e.g., carrying) to use this information.

4.4.13.2.3 Carrying

The *Carrying class* is a specialized child class of the *Task class*, designed to handle the transportation of items between specified zones. It extends the parent class by adding

carrying-specific functionality, such as QR code navigation, lifting mechanism control, and complex junction navigation. By leveraging the core functionality of the Task class—such as robot control, obstacle avoidance, and line following—the Carrying class focuses on implementing logic tailored to item transportation tasks.

4.4.13.2.3.1 Key Features and Functionality

Task Initialization:

The Carrying class initializes by accepting a list of tasks (params) that specify the zones and actions to be performed, such as loading or unloading. It calls the parent class's constructor with the task name "carrying" to set up the core functionality. This initialization ensures the robot is ready to execute the carrying task with the necessary parameters.

QR Code Navigation:

The class uses QR codes to navigate to goal positions. It implements the *_check_qr* method to detect QR codes and determine if the robot has reached a goal zone. By comparing the detected QR code with the target zone, the robot can confirm its location and proceed with the next action, such as loading or unloading.

Lifting Mechanism:

The Carrying class controls a lifting mechanism to handle items during loading and unloading. The *_lift* method is responsible for interpolating the lifting motion over a specified duration, ensuring smooth and precise movement. Commands for the lifting mechanism are published to the */position_joint_controller/command* topic, enabling real-time control.

Junction Navigation:

The class extends the *junction_decision* method to handle complex navigation decisions. When the robot encounters a junction, it determines the best direction to reach the goal zone based on QR code positions and intersections. The class uses a *shortest path algorithm* to choose between turning left, right, or making a U-turn, ensuring efficient navigation.

Task Completion:

The class tracks the progress of the task using *self._currentTask*. Once all goals have been reached, the task is marked as complete, and a signal (*fsignal*) is emitted to notify the system. This ensures that other components are aware of the task's completion and can take appropriate action.

4.4.13.2.3.2 How It Works

Initialization:

The Carrying class initializes by calling the parent class's constructor and setting up the task parameters. It prepares the robot for the carrying task by loading QR code positions and

configuring the lifting mechanism.

Navigation and QR Code Detection.
The robot navigates through the environment using QR codes as markers. The *_check_qr* method detects QR codes and determines if the robot has reached a goal zone. If the target zone is reached, the robot proceeds with loading or unloading.

Lifting Mechanism Control:
The *_lift* method controls the lifting mechanism, interpolating its motion to ensure smooth operation. This method is called during loading and unloading to handle items efficiently.

Junction Navigation:
The *junction navigation* is a critical component of the Carrying class, responsible for handling navigation decisions at junctions. Its purpose is to determine the best action—such as turning left, right, or moving forward—based on the detected QR codes, the robot's current position, and the goal zone. This method ensures the robot navigates efficiently and reaches its destination while avoiding unnecessary detours.

The method takes three key parameters: *onLeft* Indicates whether a left turn is detected, *onRight* Indicates whether a right turn is detected and *onTop* Indicates whether a straight path is detected. These parameters provide information about the robot's immediate environment, helping it decide the best course of action.

The method's logic is divided into two main scenarios: when no QR code is detected and when a QR code is detected.

No QR Code Detected: If no QR code has been scanned recently, the robot makes a decision based on the detected junctions:

- If a *left or right turn* is detected, the robot prepares to turn in the corresponding direction.
- If a *straight path* is detected, the robot moves forward to continue its journey.

This logic ensures the robot continues exploring or navigating even when no QR codes are present, maintaining progress toward its goal.

QR Code Detected: If a QR code has been scanned, the robot checks whether it corresponds to a *goal zone* or an *intersection*:

- If the QR code is a *goal zone*, the robot moves to the lifting position and performs the required action, such as loading or unloading. This marks a key milestone in the task.
- If the QR code is an *intersection*, the robot uses the *_chooseSide* or *_chooseSideToGo* method to determine the best direction to reach the goal zone. These methods calculate the optimal path based on the robot's current position, the goal zone's location, and the available routes.

This logic ensures the robot makes informed decisions at intersections, minimizing travel time and energy consumption.

Based on the decision-making process, the method publishes velocity commands (self.msg) to the */cmd_vel* topic. These commands execute the chosen action, such as turning left, turning right, or moving forward. This ensures the robot's movements are precise and aligned with the navigation strategy.

Code Block 4.4.28: Junction Decision Function Of Carrying Process

```
def junction_decision(self, onLeft, onRight, onTop):
    print("Making decision")
    print(self.hasDetectedQrRecently)
    self.stop()
    tm = 3
    current = self._params[self._currentTask]

    if not self.hasDetectedQrRecently:  # no qrCode has been scanned
        if onLeft and (not onTop) and (not onRight):
            self._turnSide = "left"
            self.moveToTurnPosition = True
        elif onRight and (not onTop) and (not onLeft):
            self._turnSide = "right"
            self.moveToTurnPosition = True
        elif onTop:
            tm = 2
            self._move_forward()
            self.timer = rospy.Timer(rospy.Duration(tm),
             ↪  self.resume_processing, oneshot=True)
        else:  # if the robot can't go straight, failed the task
            self._task_failed("The robot is stuck : should go straight
             ↪  ")

    elif self._lastQrCode:  # qrCode has been scanned
        if self._lastQrCode in self.qrcodes:
            if self._isStationSide(self._lastQrCode):
                tm = 2
                self._move_forward()
                self.timer = rospy.Timer(rospy.Duration(tm),
                 ↪  self.resume_processing, oneshot=True)
            else:
                if self.goal_in_db:
```

```
            current = self._params[self._currentTask]
            zArr = current["zone"].split(" ")
            goal = f"{zArr[0]}_{zArr[1]}_center"
            goalPos = self.qrcodes.get(goal, (None, None))
            side = self._chooseSideToGo(onTop, onRight, onLeft,
            ↪  goalPos)
            if side == "S":
                tm = 2
                self._move_forward()
                self.timer = rospy.Timer(rospy.Duration(tm),
                ↪  self.resume_processing, oneshot=True)
            elif side == "R":
                self._turnSide = "right"
                self.moveToTurnPosition = True
            elif side == "L":
                self._turnSide = "left"
                self.moveToTurnPosition = True
            elif side == "B":
                tm = 6
                self._U_turn()
                self.timer = rospy.Timer(rospy.Duration(tm),
                ↪  self.resume_processing, oneshot=True)
            elif side == "0":
                return
        else:
            intersections =
            ↪  self._getIntersection(current['zone'])
            if self._lastQrCode in [intersections[0][0],
            ↪  intersections[1][0]]:
                if onLeft:
                    self._turnSide = "left"
                    self.moveToTurnPosition = True
                elif onRight:
                    self._turnSide = "right"
                    self.moveToTurnPosition = True
                else:
                    self._task_failed("
                    ↪  The robot is stuck : should turn")
            else:
                if onLeft and (not onTop) and (not onRight):
```

```
                    self._turnSide = "left"
                    self.moveToTurnPosition = True
                elif onRight and (not onTop) and (not onLeft):
                    self._turnSide = "right"
                    self.moveToTurnPosition = True
                else:
                    print("Not a corner, using shortest path")
                    side = self._chooseSide(onTop, onRight,
                     ↪  onLeft, intersections)
                    if side == "S":
                        tm = 2
                        self._move_forward()
                        self.timer =
                         ↪  rospy.Timer(rospy.Duration(tm),
                         ↪  self.resume_processing,
                         ↪  oneshot=True)
                    elif side == "R":
                        self._turnSide = "right"
                        self.moveToTurnPosition = True
                    elif side == "L":
                        self._turnSide = "left"
                        self.moveToTurnPosition = True
                    elif side == "B":
                        tm = 6
                        self._U_turn()
                        self.timer =
                         ↪  rospy.Timer(rospy.Duration(tm),
                         ↪  self.resume_processing,
                         ↪  oneshot=True)
                    elif side == "0":
                        tm = 6
                        self._U_turn()
                        self.timer =
                         ↪  rospy.Timer(rospy.Duration(tm),
                         ↪  self.resume_processing,
                         ↪  oneshot=True)
                        rospy.logerr("Wrong direction")
                        return
            else:
                tm = 2
```

```
            self._move_forward()
            self.timer = rospy.Timer(rospy.Duration(tm),
             ↪  self.resume_processing, oneshot=True)
            rospy.logerr("The robot is stuck : doesn't know what to do
             ↪  ")
```

Task Completion: Once all goals have been reached, the task is marked as complete, and the *fsignal* is emitted to notify the system. The parent class's cleanup methods are called to ensure resources are released and the system is ready for the next task.

4.4.13.2.3.3 Interaction Between Methods

- *Junction Detection*: The junction_decision method detects junctions and determines if the robot should turn left, right, or move forward. If a QR code is detected, it uses _chooseSide or _chooseSideToGo to determine the best direction to reach the goal zone.
- *Path Planning*: The _chooseSide method calculates the shortest path to the goal zone based on the positions of intersections. The _chooseSideToGo method calculates the shortest path to a specific target position (e.g., a goal zone).
- *Navigation*:
  - Based on the chosen direction, the robot executes the corresponding action (e.g., turn left, turn right, move forward).

4.4.13.2.3.4 Example Scenario

1. Approaching a Junction: The robot detects a junction with options to turn left, right, or move forward. The *junction_decision* method calls *_chooseSide* to determine the best direction to reach the goal zone.
2. Calculating the Shortest Path: The *_chooseSide* method calculates the Euclidean distance to each intersection and chooses the direction with the minimum distance. If the goal zone is directly ahead, the robot moves forward. If it's to the left or right, the robot turns accordingly.
3. Executing the Action: The robot executes the chosen action (e.g., turn left, turn right, move forward) and continues navigating toward the goal zone.

4.4.13.2.3.5 Impact on the System

- These methods enable the robot to *autonomously navigate junctions* and *reach goal zones efficiently*, ensuring reliable and efficient task execution.
- They demonstrate the importance of *path planning* and *decision-making algorithms* in robotics, showcasing your ability to implement complex navigation logic.

Code Block 4.4.29: Adjust Orientation Function of Carrying Class

```
def _adjust_orientation(self, error, angle, pixels, mask, w_min,
 ↪  h_min):

        if not self.isLifting :
            if self.moveToLiftPosition:
                if self._distanceToLifePosition() > 0.005:
                    self._move(error)
                    return None
                else:
                    print("Lift position reached")
                    self.stop()
                    self.isLifting = True
                    self._lift()
                return None
            else:
                return super()._adjust_orientation(error, angle,
                 ↪  pixels, mask, w_min, h_min)
```

Code Block 4.4.30: Lift function

```
def _lift(self, duration=3):
    current = self._params[self._currentTask]
    liftType = current["type"]
    start = 0 if liftType == "Loading" else 1
    end = 1 if liftType == "Loading" else 0

    start_time = rospy.Time.now()
    end_time = start_time + rospy.Duration.from_sec(duration)
    rate = rospy.Rate(100)
    while rospy.Time.now() < end_time and not rospy.is_shutdown():
        current_time = rospy.Time.now()
        elapsed = (current_time - start_time).to_sec()
```

```
        fraction = elapsed / duration
        fraction = min(fraction, 1.0)  # Fixed the missing parenthesis
        ↪ here
        current_value = start + fraction * (end - start)
        self.liftPub.publish(current_value)
        rate.sleep()

    self.liftPub.publish(end)
    self.isLifting = False
    self.moveToLiftPosition = False

    self._currentTask += 1
    if(self._currentTask == len(self._params)):
        self.stop()
        self._task_finished(message="All goals have been reached")
```

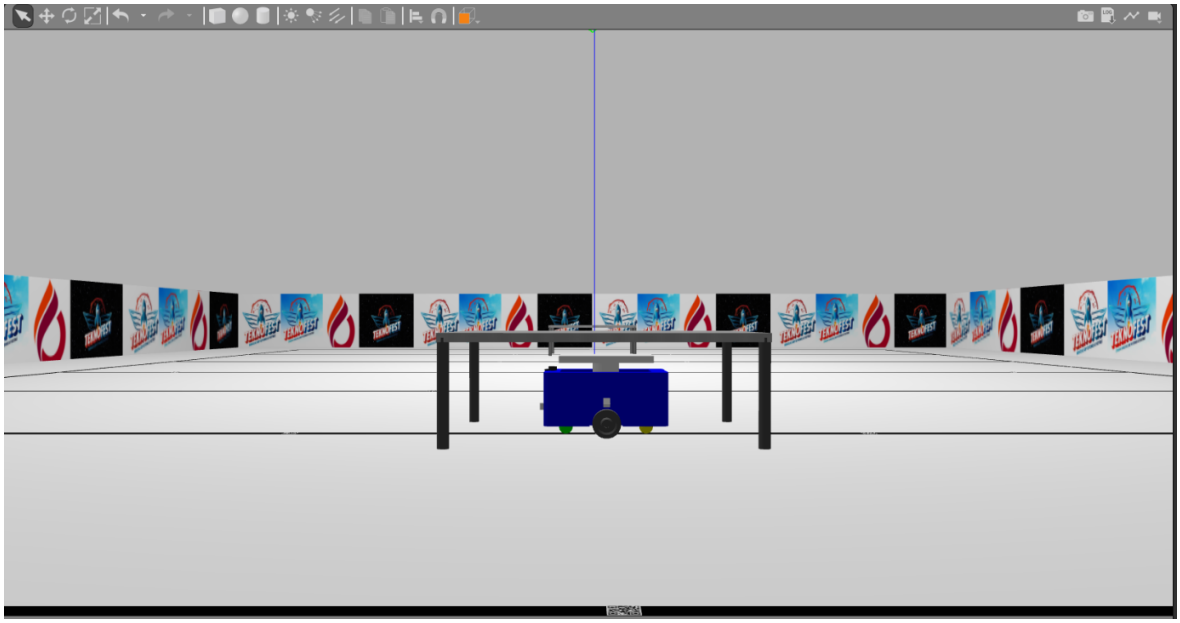


Figure 4.102: the robot ready to lift a charge

CHAPTER FIVE
CONCLUSION AND FUTURE WORKS

This project has successfully addressed the design and optimization of critical components for automated systems, focusing on bearing wheels, caster wheels, and their integration into advanced material handling solutions such as AGVs (Automated Guided Vehicles). The study employed a systematic approach to ensure structural integrity, operational efficiency, and compliance with industry standards. Key achievements include the validation of the single-level scissor lift mechanism for a load capacity of 200 kg, the development of a robust chassis structure capable of supporting up to 300 kg (2943N), and the selection of the NSK 6201 bearing for its high performance and reliability in handling both radial and axial loads. SolidWorks was utilized for 3D modeling and finite element analysis (FEA), enabling accurate identification of potential failure points and enhancing structural integrity while minimizing prototyping costs. Through careful consideration of material properties, mechanical constraints, and environmental factors, the final designs demonstrate excellent durability, efficiency, and compliance with ISO 3691-4:2020 guidelines.

5.1 METHODOLOGY AND DESIGN APPROACH

The design process adopted a comprehensive methodology that combined theoretical calculations, empirical analysis, and digital tools. Extensive structural analysis and stress testing were conducted to validate the performance of the scissor lift mechanism and chassis, ensuring that the designs met specified operational requirements while maintaining necessary safety margins. SolidWorks simulations played a pivotal role in refining the design. FEA helped identify weak points in the assemblies, allowing for iterative improvements before physical prototyping. This approach not only enhanced the efficiency of the design process but also reduced costs significantly. Adherence to ISO 3691-4:2020 guidelines ensured that the bearing and caster wheels met stringent safety requirements, including braking systems, speed control, load handling, and stability. Compliance with these standards guarantees safe and efficient operation in industrial environments.

5.2 CONTRIBUTIONS TO INDUSTRIAL APPLICATIONS

The successful development of these components represents a significant advancement in automated material handling solutions. By integrating advanced safety features, control systems, and predictive maintenance capabilities, the system demonstrates improved reliability and functionality. Enhanced operational stability and structural integrity were achieved through meticulous load distribution and stress analysis. Improved load-handling capabilities for industrial trucks were realized, aligning with modern trends toward automation and sustainability. These contributions position the system as a reliable solution for industrial applications, addressing the growing demand for efficient and durable material handling equipment.

5.3 FUTURE DIRECTIONS

While the current project lays a strong foundation, several areas have been identified for future development to further enhance performance, safety, and sustainability. Future work will focus on optimizing the weight and strength of components by exploring lightweight materials. Modular attachment systems and enhanced payload configurations will also be investigated to improve versatility. The integration of smart sensors for real-time load monitoring and predictive maintenance is a priority, enabling proactive maintenance, reducing downtime, and extending the service life of components. Advanced emergency stop systems and collision avoidance capabilities will be developed to ensure the AGV system adapts to complex industrial environments while maintaining high safety standards. Exploring eco-friendly material alternatives aligns with global environmental regulations and industry trends, reducing the carbon footprint while enhancing durability and recyclability. Experimental testing and field applications are essential to validate the proposed designs in real-world scenarios, providing valuable insights into performance under varying conditions and helping refine the system further.

5.4 CONCLUSION

In conclusion, this project has demonstrated a structured and comprehensive approach to the design and analysis of bearing wheels, caster wheels, and AGV systems. By leveraging advanced tools like SolidWorks for simulation and adhering to industry standards such as ISO 3691-4:2020, the study provides a robust framework for developing high-performance, durable components. The successful implementation of the proposed designs marks a significant step forward in automated material handling solutions. With continued advancements

in materials, smart technologies, and sustainable practices, these systems will evolve to meet emerging industrial demands, offering greater efficiency, versatility, and safety. Ultimately, this project paves the way for more reliable and innovative solutions in industrial automation and material transport.

APPENDIX

Bearing Specifications

- Bearing Type: Deep Groove Ball Bearing (NSK 6201)
- Inner Diameter: 12 mm
- Outer Diameter: 32 mm
- Load Capacity: 7.28 kN

Material Properties

- AISI 1045 Steel: Yield Strength = 530 MPa, Ultimate Tensile Strength = 625 MPa
- AISI 52100 Chrome Steel: High Hardness (Rockwell C 60-67), Wear Resistance

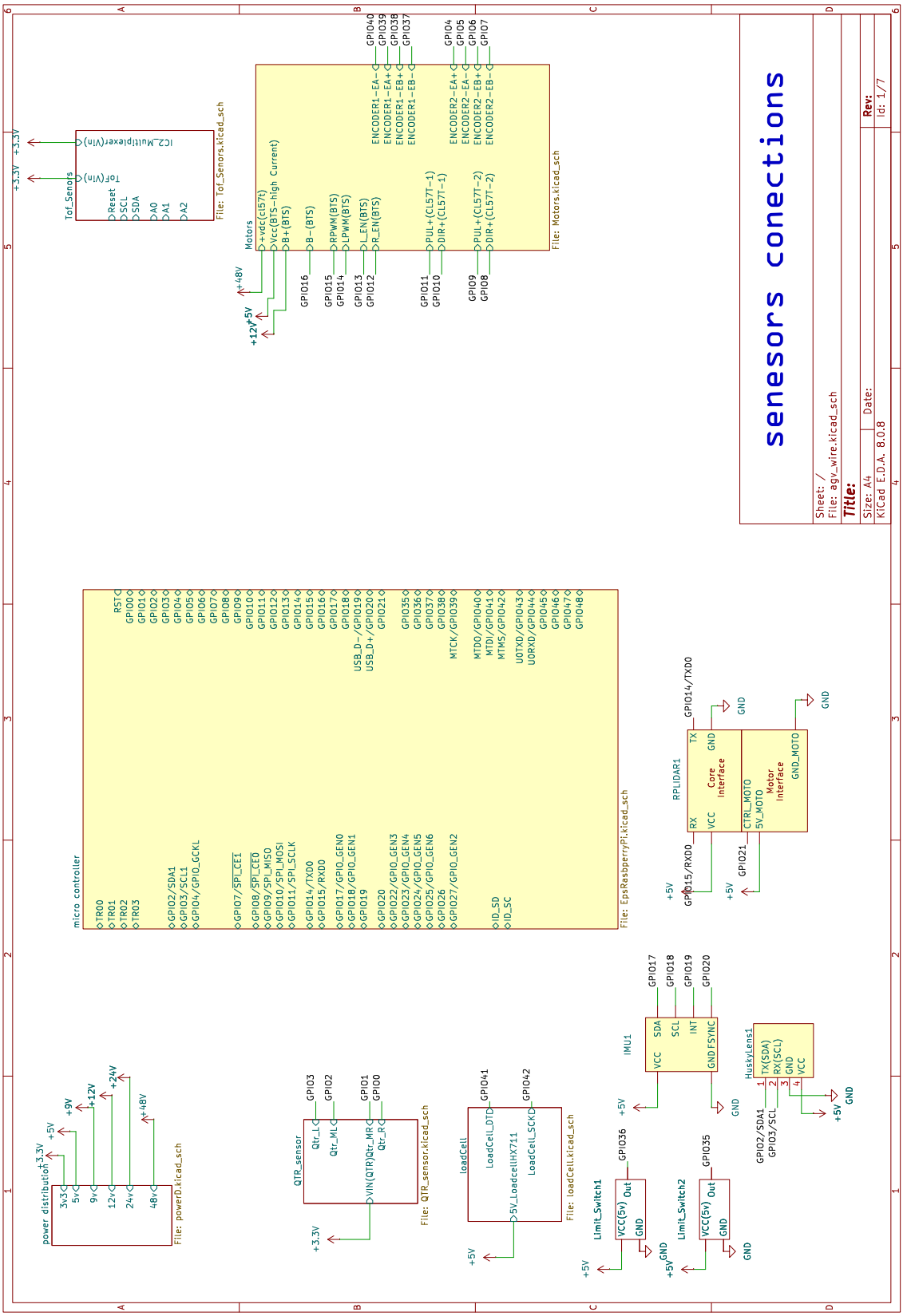


Figure 1: AGV Wire Diagram (Full Page)

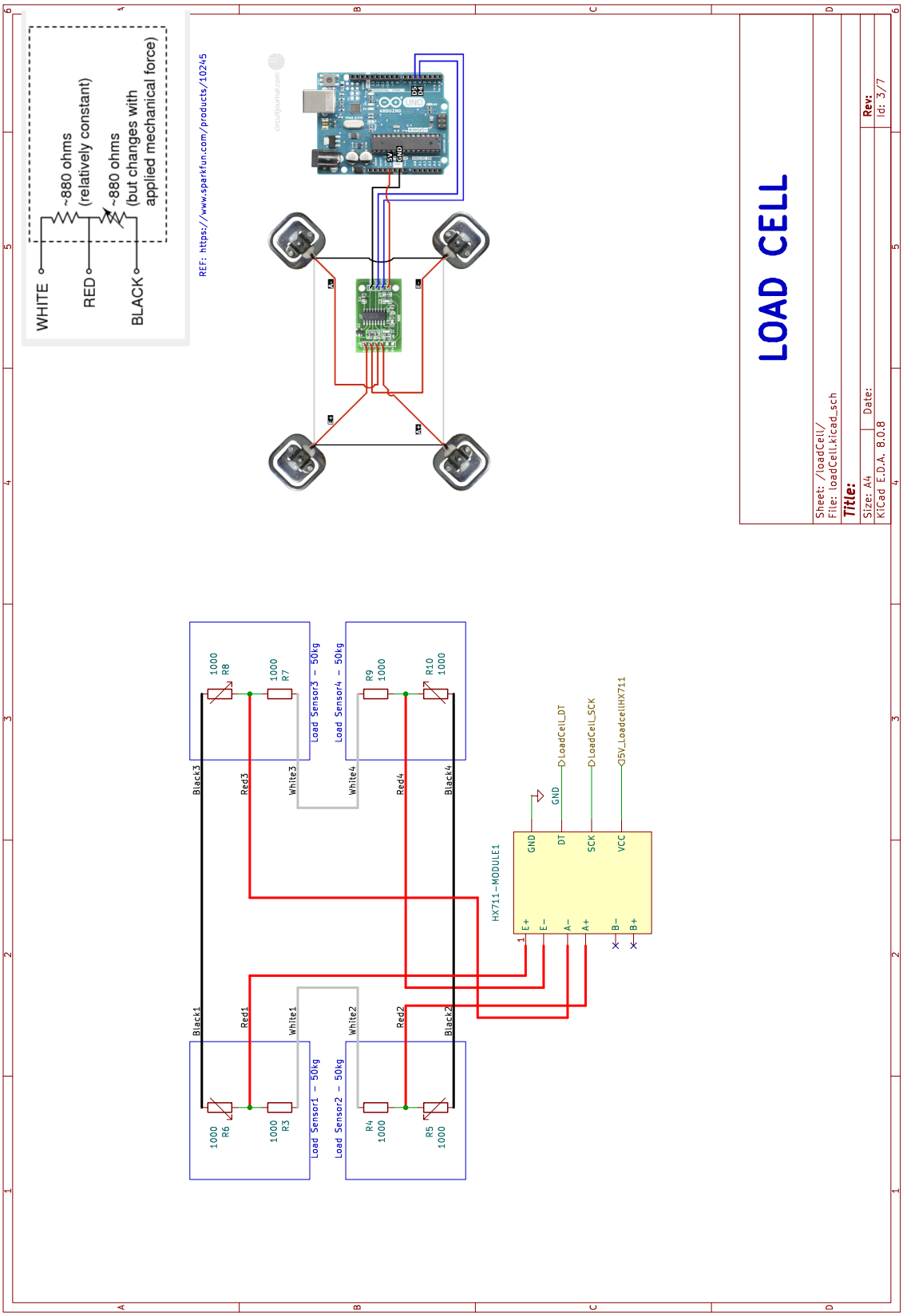


Figure 2: loadCell Circuit

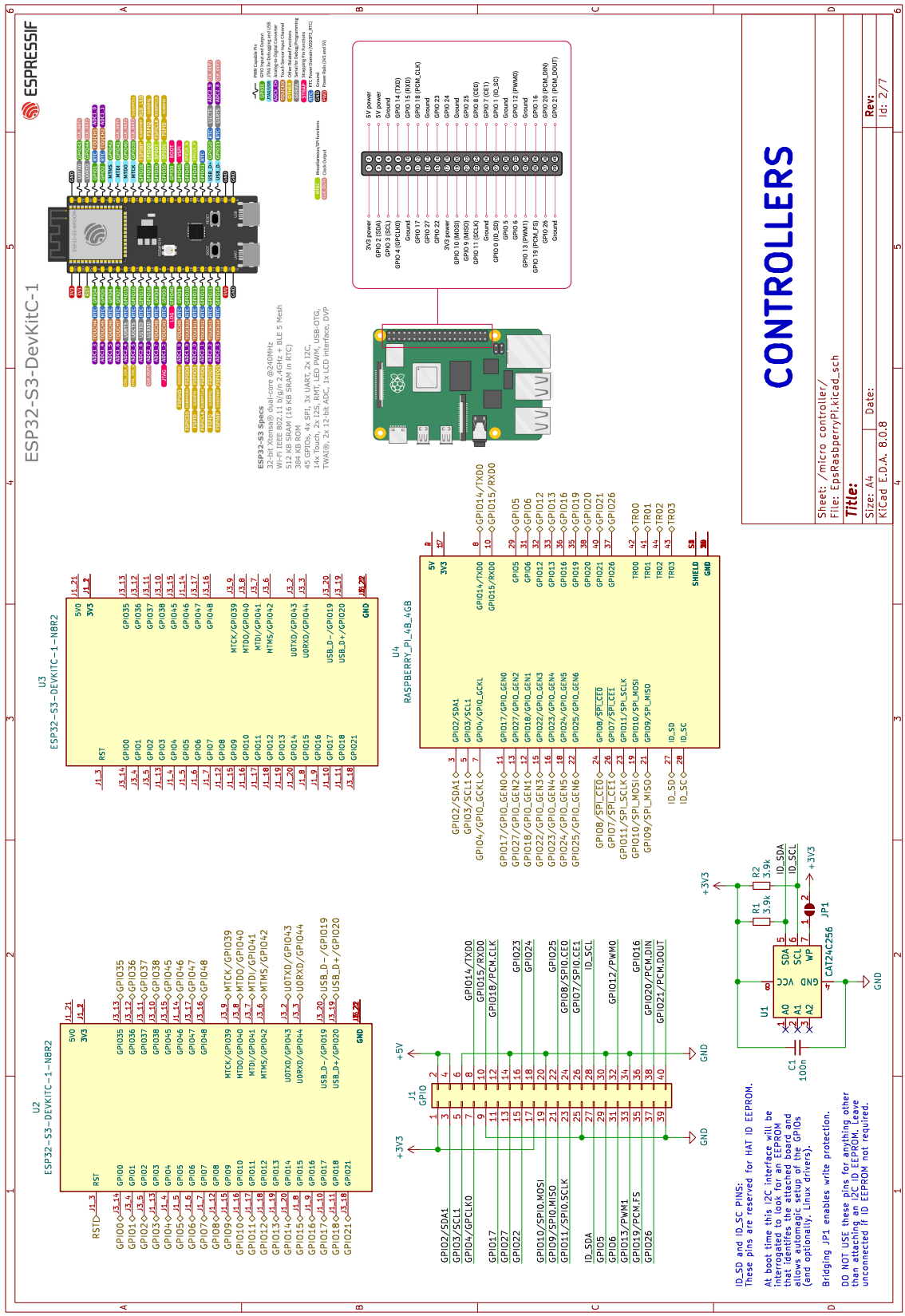


Figure 3: Microcontrollers connections

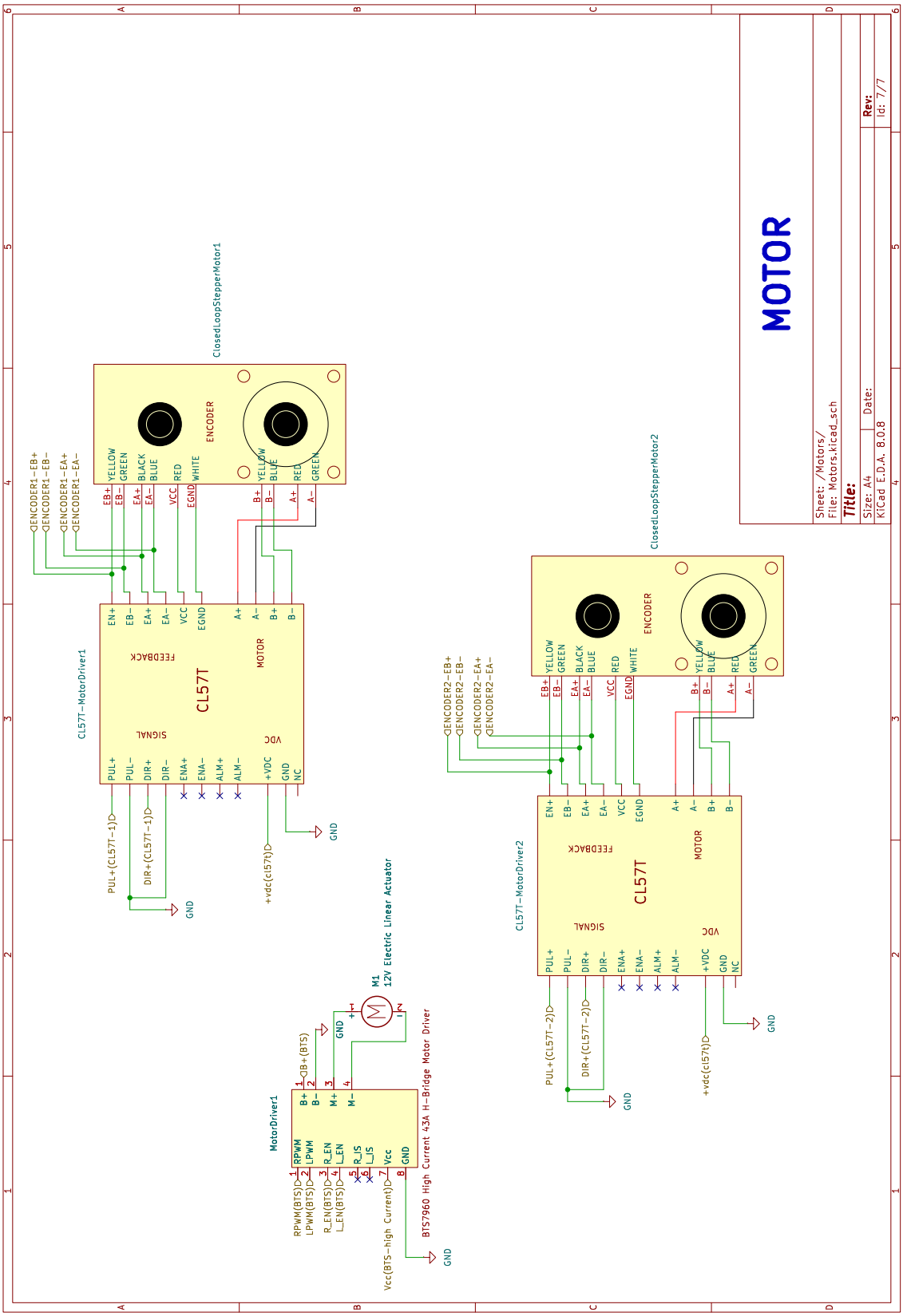


Figure 4: AGV Motors

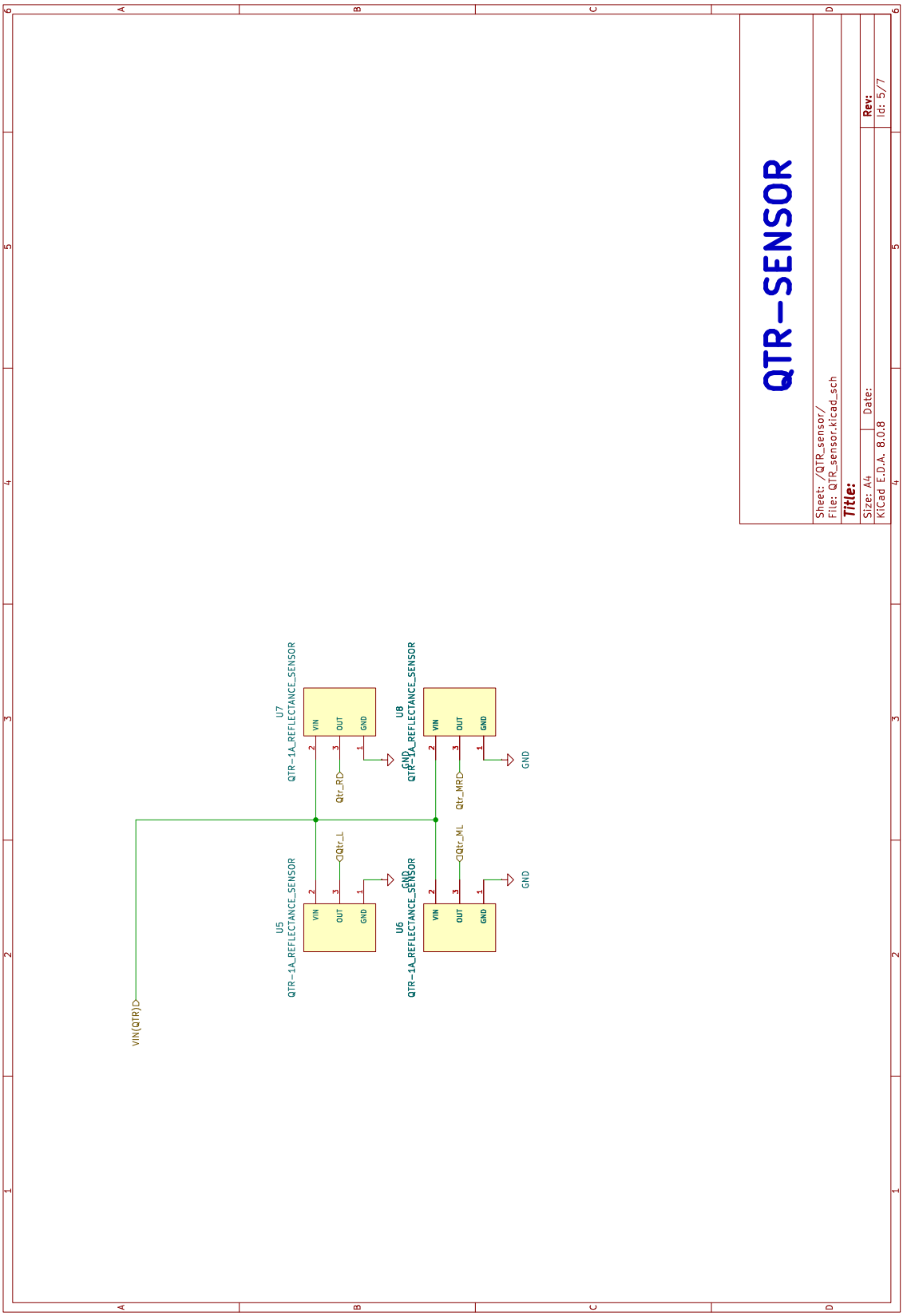


Figure 5: QTR Sensors

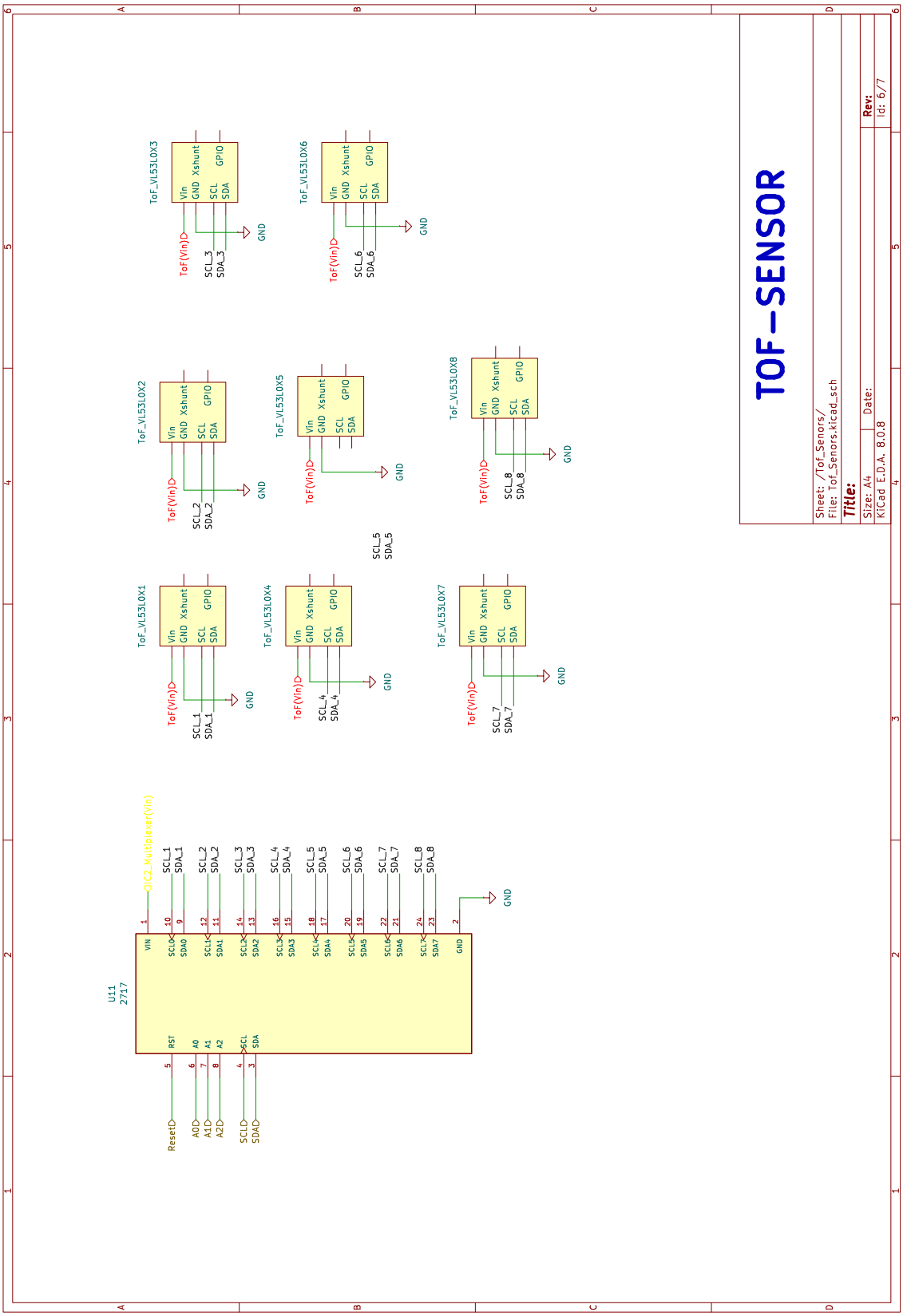


Figure 6: ToF Sensors

Table 1: List of Useful ROS Packages for Robotics Development

Category	Package Name	Description
URDF	urdf	Core package for defining robots in URDF format.
URDF	xacro	XML macro language for simplifying URDF files.
URDF	robot_state_publisher	Publishes the state of the robot to tf.
URDF	joint_state_publisher	Publishes joint states for simulating joint movements.
URDF	kdl_parser	Parses URDF into KDL trees for kinematic calculations.
URDF	srdf	Semantic Robot Description Format for MoveIt!.
Gazebo	gazebo_ros_pkgs	ROS integration with Gazebo.
Gazebo	gazebo_plugins	Plugins for sensors like cameras, lasers, and IMUs.
Gazebo	gazebo_ros_control	Integrates ROS control with Gazebo.
Gazebo	hector_gazebo_plugins	Additional plugins for GPS, sonar, and IMU sensors.
Gazebo	ros_control	Framework for controlling robots in simulation and real hardware.
Gazebo	controller_manager	Manages controllers for joints in Gazebo.
Gazebo	effort_controllers	Effort-based controllers for joints.
Gazebo	position_controllers	Position-based controllers for joints.
Gazebo	velocity_controllers	Velocity-based controllers for joints.
Navigation	move_base	Core package for navigation stack, responsible for global and local path planning.

Continued on next page

Table 1 – continued from previous page

Category	Package Name	Description
Navigation	amcl	Adaptive Monte Carlo Localization for 2D pose estimation.
Navigation	gmapping	SLAM algorithm using laser scans.
Navigation	cartographer	Advanced SLAM library supporting 2D and 3D mapping.
Navigation	navigation	Full ROS Navigation stack.
Navigation	global_planner	Global path planner (e.g., A*, Dijkstra).
Navigation	local_planner	Local path planner (e.g., DWA, TEB).
Navigation	teb_local_planner	Timed Elastic Band local planner for dynamic environments.
Navigation	costmap_2d	2D costmap representation for obstacle avoidance.
Navigation	dwa_local_planner	Dynamic Window Approach for local path planning.
Navigation	nav_msgs	Messages related to navigation (e.g., Odometry, Path).
Communication	rospy	Python client library for ROS.
Communication	roscpp	C++ client library for ROS.
Communication	std_msgs	Standard message types (e.g., Float32, String, Bool).
Communication	sensor_msgs	Messages for sensor data (e.g., LaserScan, Image, Imu).
Communication	geometry_msgs	Messages for geometry-related data (e.g., Pose, Twist, Point).
Communication	tf / tf2	Transform library for managing coordinate frames.
Communication	actionlib	Action server/client for long-running tasks.
Communication	message_filters	Synchronizes multiple topics based on timestamps.
Communication	rosserial	Communicate with microcontrollers over serial.

Continued on next page

Table 1 – continued from previous page

Category	Package Name	Description
Communication	rosbridge_suite	WebSocket-based communication for web-based interfaces.
Communication	rosapi	Provides REST API access to ROS topics, services, and parameters.
Manipulation	moveit	Motion planning framework for robotic arms.
Manipulation	moveit_core	Core components of MoveIt! (planning, kinematics, etc.).
Manipulation	moveit_ros	ROS integration for MoveIt!.
Manipulation	moveit_commander	Python interface for MoveIt!.
Manipulation	trac_ik	Inverse Kinematics solver that works well with MoveIt!.
Manipulation	moveit_visual_tools	Tools for visualizing motion planning in RViz.
Manipulation	grasp_generator	Generates grasps for robotic manipulators.
Visualization	rviz	3D visualization tool for ROS.
Visualization	rviz_plugin_tutorials	Tutorials for creating custom RViz plugins.
Visualization	rviz_imu_plugin	Plugin for visualizing IMU data in RViz.
Visualization	rviz_satellite	Plugin for visualizing satellite imagery in RViz.
Sensors	laser_geometry	Converts laser scans into point clouds.
Sensors	image_transport	Handles image transport (e.g., compressed images).
Sensors	camera_info_manager	Manages camera calibration information.
Sensors	depth_image_proc	Processes depth images (e.g., from RGB-D cameras).
Sensors	pcl_ros	ROS integration with Point Cloud Library (PCL) for 3D perception.

Continued on next page

Table 1 – continued from previous page

Category	Package Name	Description
Sensors	octomap	3D occupancy grid mapping for collision avoidance.
Sensors	rtabmap_ros	Real-Time Appearance-Based Mapping for SLAM.
Sensors	aruco_ros	Detects ArUco markers for augmented reality and localization.
Control	control_toolbox	Provides PID controllers and other control utilities.
Control	realtime_tools	Tools for real-time control in ROS.
Control	trajectory_msgs	Messages for defining trajectories (e.g., JointTrajectory).
Control	joint_trajectory_controller	Controller for executing joint trajectories.
Control	gazebo_ros_force_system	Applies forces to objects in Gazebo.
Other	dynamic_reconfigure	Allows dynamic reconfiguration of node parameters at runtime.
Other	diagnostic_updater	Monitors the health of ROS nodes and publishes diagnostic messages.
Other	robot_localization	State estimation package that fuses data from multiple sensors.
Other	slam_toolbox	Flexible SLAM solution with support for online and offline mapping.
Other	rosbag	Records and plays back ROS messages for debugging and testing.
Other	rosparam_shortcuts	Simplifies loading parameters from the parameter server.

BIBLIOGRAPHY

[1] L. Joseph and J. Cacace, Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System. Packt Publishing Ltd, 2018.

[2] S. K. Das, "Design and methodology of line follower automated guided vehicle-a review," International Journal of Science Technology & Engineering, vol. 2, no. 10, pp. 9–13, 2016.

[3] A. J. Moshayedi, L. Jinsong, and L. Liao, "Agv (automated guided vehicle) robot: Mission and obstacles in design and performance," Journal of Simulation and Analysis of Novel Technologies in Mechanical Engineering, vol. 12, no. 4, pp. 5–18, 2019.

[4] S. A. Reveliotis, "Conflict resolution in agv systems," Iie Transactions, vol. 32, no. 7, pp. 647–659, 2000.

[5] L. Shengfang and H. Xingzhe, "Research on the agv based robot system used in substation inspection," in 2006 International Conference on Power System Technology, pp. 1–4, IEEE, 2006.

[6] International Organization for Standardization, "Industrial Trucks—Safety Requirements and Verification—Part 4: Driverless Industrial Trucks and Their Systems." ISO 3691-4:2020, 2020.

[7] American National Standards Institute (ANSI), "ANSI/ITSDF B56.5: Safety Standard for Driverless Automated Industrial Trucks." ANSI/ITSDF B56.5, 2020.

[8] European Committee for Standardization (CEN), "EN 3691-4: Safety Requirements for Industrial Trucks – Part 4: Driverless Industrial Trucks and Their Systems." EN 3691-4, 2020.

[9] Robotic Industries Association (RIA), "RIA R15.08: Industrial Robot Safety Standard." RIA R15.08, 2020.

[10] Verein Deutscher Ingenieure (VDI), "VDI 2510: Safety of Automated Systems." VDI Guideline 2510, 2017.

[11] Occupational Safety and Health Administration (OSHA), “OSHA Requirements for Industrial Safety.” OSHA Regulations, 2020.

[12] European Union, “General Data Protection Regulation (GDPR).” EU Regulation 2016/679, 2018.

[13] California State Legislature, “California Consumer Privacy Act (CCPA).” California Civil Code 1798.100 et seq., 2018.

[14] M. P. Groover, Automation, production systems, and computer-integrated manufacturing. Pearson Education India, 2016.

[15] R. N. Jazar and R. N. Jazar, “Road dynamics,” Advanced Vehicle Dynamics, pp. 297–350, 2019.

[16] M. F. Ashby, “Materials selection in mechanical design,” Metallurgia Italiana, vol. 86, pp. 475–475, 1994.

[17] F. ., S. Alhady, and W. Rahiman, “A review of controller approach for autonomous guided vehicle system,” Indonesian Journal of Electrical Engineering and Computer Science, vol. 20, pp. 552–562, 10 2020.

[18] Á. Cservenák, “Further development of an agv control system,” in Vehicle and Automotive Engineering 2: Proceedings of the 2nd VAE2018, Miskolc, Hungary, pp. 376–384, Springer, 2018.

[19] O. Motor, “Brushless dc motors for agv designs,” 2023. Accessed: October 15, 2023.

[20] L. Engineering, “Autonomous guided vehicles (agv),” 2024. Accessed: 2025-02-07.

[21] A. M. Controls, “Servo drives for agvs: Top 5 benefits,” 2024. Accessed: 2025-02-07.

[22] ROS Community, “Robot operating system (ros).” `https://www.ros.org/`, 2025. Accessed: 2023-10-12.

[23] R. Wiki, “urdf/xml - ros wiki,” 2023.

[24] N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), (Sendai, Japan), pp. 2149–2154, IEEE, Sep 2004.

[25] R. L. Smith, “Open dynamics engine.” `https://bitbucket.org/odedevs/ode/`, 2016. Online.

[26] R.-T. P. Simulation, "Bullet physics library." http://bulletphysics.org/wordpress/, 2016. Online.

[27] "Simbody: Multibody physics api." https://simtk.org/home/simbody/, 2016. Online.

[28] G. T. G. Lab and H. R. Lab, "Dart (dynamic animation and robotics toolkit)." http://dartsim.github.io/, 2016. Online.

[29] OGRE3D, "Object-oriented graphics rendering engine." http://www.ogre3d.org/, 2016. Online.

[30] F. R. Lera, F. C. Garcia, G. Esteban, and V. Matellan, "Mobile robot performance in robotics challenges: Analyzing a simulated indoor scenario and its translation to real-world," in Proc. 2014 Second International Conference on Artificial Intelligence, Modelling and Simulation, pp. 149–154, 2014.

[31] M. P. Groover, "Fundamentals of modern manufacturing materials," 2019.

[32] P. F. Brown, S. A. Della Pietra, V. J. Della Pietra, and R. L. Mercer, "The mathematics of statistical machine translation: Parameter estimation," Computational linguistics, vol. 19, no. 2, pp. 263–311, 1993.

[33] F. Can and E. A. Ozkarahan, "Concepts and effectiveness of the cover-coefficient-based clustering methodology for text databases," ACM Transactions on Database Systems (TODS), vol. 15, no. 4, pp. 483–517, 1990.

[34] METTLER TOLEDO Group, "Mettler toledo group." Accessed: [Insert Access Date if needed].

[35] E. Zainescu, "Calculation of single level scissor lift," 2019.

[36] TIMKEN, "Deep groove ball bearing catalog." Accessed: [Insert Access Date if needed].

[37] R. C. Hibbeler, Engineering Mechanics Statics and Dynamics. fourteenth edition ed., 2016.

[38] MCLE 476 Lectures, "Introduction mechanisms and kinematics." Accessed: [Insert Access Date if needed].

[39] Eagle, "Aluminum catalogs." Accessed: [Insert Access Date if needed].

[40] P. D. Smith and J. R. Wilson, "Design and analysis of automated guided vehicles," Journal of Industrial Engineering, vol. 45, no. 3, pp. 156–172, 2024.

[41] R. Kumar, "Structural analysis using solidworks simulation," in Mechanical Design Engineering Handbook, Academic Press, 2nd edition ed., 2024.

[42] M. Thompson, "Advanced scissor lift mechanisms: Design and implementation," International Journal of Mechanical Engineering, vol. 12, no. 4, pp. 78–92, 2023.

[43] H. Chen and L. Zhang, "Load analysis in agv chassis design," Robotics and Automation Systems, vol. 33, pp. 245–260, 2024.

[44] B. Anderson, "Material selection for industrial agv applications," Materials Today: Proceedings, vol. 28, pp. 1123–1135, 2024.

[45] K. Roberts, Safety Standards in Automated Material Handling Systems. Springer, 5th edition ed., 2023.

[46] D. Williams and S. Brown, "Stress analysis techniques in manufacturing design," Manufacturing Technology Today, vol. 15, no. 2, pp. 112–128, 2024.

[47] T. Johnson, "Modern agv systems: Design principles and applications," Automation Engineering Review, vol. 8, no. 1, pp. 45–62, 2024.