Part I

INTRODUCTION

CYPRUS INTERNATIONAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

DIGITAL TECHNOLOGIES IN INDUSTRY

By

HASHEM VASEGHI

PARFAIT ZAINA NGOI

FELLY NGOY

JUDE KABEMBA

BOIMA FAHNBULLEH

GREGORY MWEMA

OLUTOYE OPEYEMI

Date: February 5, 2025

Place: Nicosia, NORTH CYPRUS

CYPRUS INTERNATIONAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

DIGITAL TECHNOLOGIES IN INDUSTRY

By

HASHEM VASEGHI

PARFAIT ZAINA NGOI

FELLY NGOY

JUDE KABEMBA

BOIMA FAHNBULLEH

GREGORY MWEMA

OLUTOYE OPEYEMI

Date: February 5, 2025

Place: Nicosia, NORTH CYPRUS

DIGITAL TECHNOLOGIES IN INDUSTRY

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.

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.

Contents

List of Figures

List of Tables

Chapter 1

introduction

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. 1.1) 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].

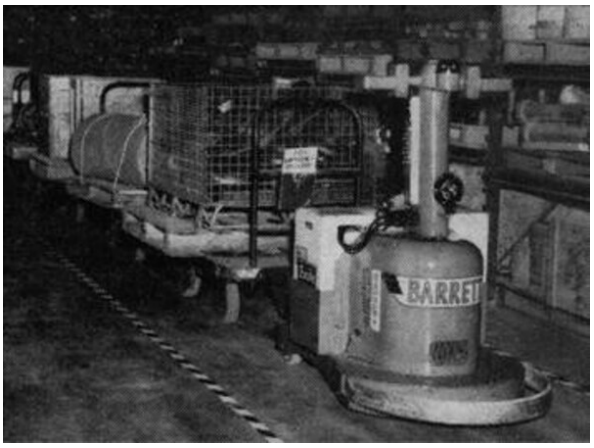


Figure 1.1: one of the Old AGV picture

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 human 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.

Chapter 2

Theory of AGV Design

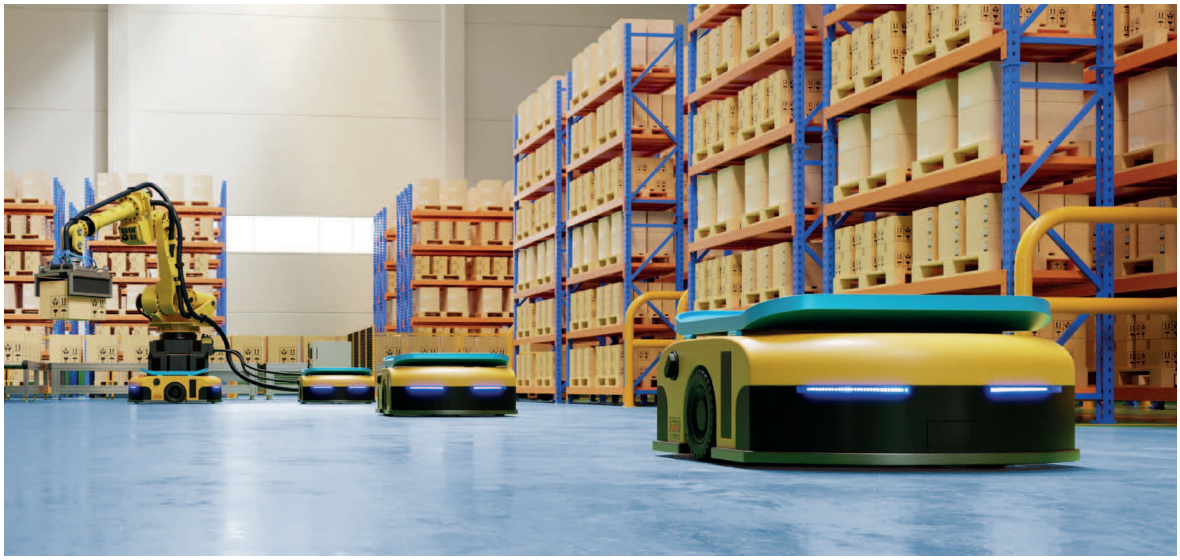


Figure 2.1: Smart AGV in Industry 4.0

AGVs are essential components in modern industrial automation systems, designed to improve efficiency, flexibility, and safety in material handling and logistics. This section discusses the fundamental theories and principles underlying AGV design, including navigation, control, and system integration.

Part II

REALISTIC CONSTRAINTS

Chapter 3

GEAR BOX DESIGN

The objective of this part of the project was to design a couple of rugged, efficient, and lightweight middle drive wheels integrated with a gearbox that can provide the needed torque and speed for the AGV.

Key objectives included:

1. To design the system in SolidWorks.
2. Check for structural integrity at operational loads.
3. Analyze gearbox performance for the required torque and speed to achieve a gear ratio of 20:1.
4. Simplify design for manufacturability and assembly.

3.1 LITERATURE SURVAY

AGVs are driverless transport systems used in the manufacturing industry, warehouses, and logistics. According to Groover (2015), 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, ”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, 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.

3.2 REALISTIC CONSTRAINTS

In designing the two middle drive wheels with a gearbox in CAD, (SolidWorks), considerations to realistic constraints must be made to ensure the design is practical, manufacturable, and functional. Many a time, these constraints are classified into technical, material, economic, environmental, and human factors.

3.3 DESIGN CONSTRAINTS

3.3.1 Technical Constraints

These refer to limitations pertaining to the design's feasibility and functionality.

Dimensional Accuracy: The dimensions need to be precise so that the fitted parts inside assemble well.

Load Bearing Capacity: The wheels and gearbox should bear the expected loads without deforming or failing.

Speed and Torque Requirements: The gear ratio and wheel design must be such that desired performance is achieved.

Compatibility with Other Components: Ensure that the design fits well with the AGV chassis, sensors, and other systems.

Assembly and Maintenance: The design must ensure ease assembly, disassembly, and maintenance.

Simulation Accuracy: Limitations of CAD software may affect the accuracy of stress, motion, or thermal simulations.

3.3.2 Material Constraints

The choice of materials impacts weight, durability, and cost.

Strength and Durability: Materials must be able to handle the mechanical stresses without failure.

Weight: Lighter materials are always preferred for efficiency, but without sacrificing strength.

Availability: Materials selected must be available to the required quantity.

Cost: The high-performance material may be beyond the budget and therefore some compromise may be necessary.

3.3.3 Manufacturing Constraints

The design should consider the limitations of manufacturing methods.

Process Feasibility: The parts must be manufacturable using available processes such as CNC machining, casting or 3D printing.

Tool Accessibility: Shapes that include internal grooves or undercuts may not easily be machined due to complexity.

Surface Finish: Very complicated surfaces may be expensive or time-consuming to produce.

Assembly Difficulty: Designs that include many parts or tight clearances may be more difficult to assemble because of mating with specific part which may not be possible if there are too tight.

Standard Components: Using off-the-shelf components such as bearings and shafts rather than customized components simplifies production and reduces cost.

3.3.4 Economic Constraints

The design should be within the financial scope of the project.

Material Cost: High-quality materials can also raise costs.

Manufacturing Cost: Complex geometries and tight tolerances raise production costs.

Time Constraints: Designs, prototyping, and testing may be too time-consuming and hence may require simplifications.

3.3.5 Environmental Constraints

The design must ensure a minimal impact on the environment.

Energy Efficiency: The system shall minimize energy losses, for example, frictional losses or heat dissipation.

Material Sustainability: Materials used should be recyclable or eco-friendly wherever possible.

Waste Management: Waste generated in manufacturing is minimized.

Operating Environment: The system should be able to put up with all forms of environmental conditions, including humidity, dust, or an extreme temperature.

3.3.6 Human Constraints

The human factors bear on the usability and safety.

Ergonomics: The system shall be easy to handle during assembly and maintenance.

Safety: Ensure the design minimizes sharp edges or exposed moving parts that could cause injury.

User Expertise: The design shall take into consideration the expertise of the operators or assemblers.

Documentation: Clear assembly and operating instructions are essential for effective use.

3.4 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 SolidWorks.
3. Simulation: motion analysis to validate the design.
4. Optimization: Refined design for performance and manufacturability.

3.4.1 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

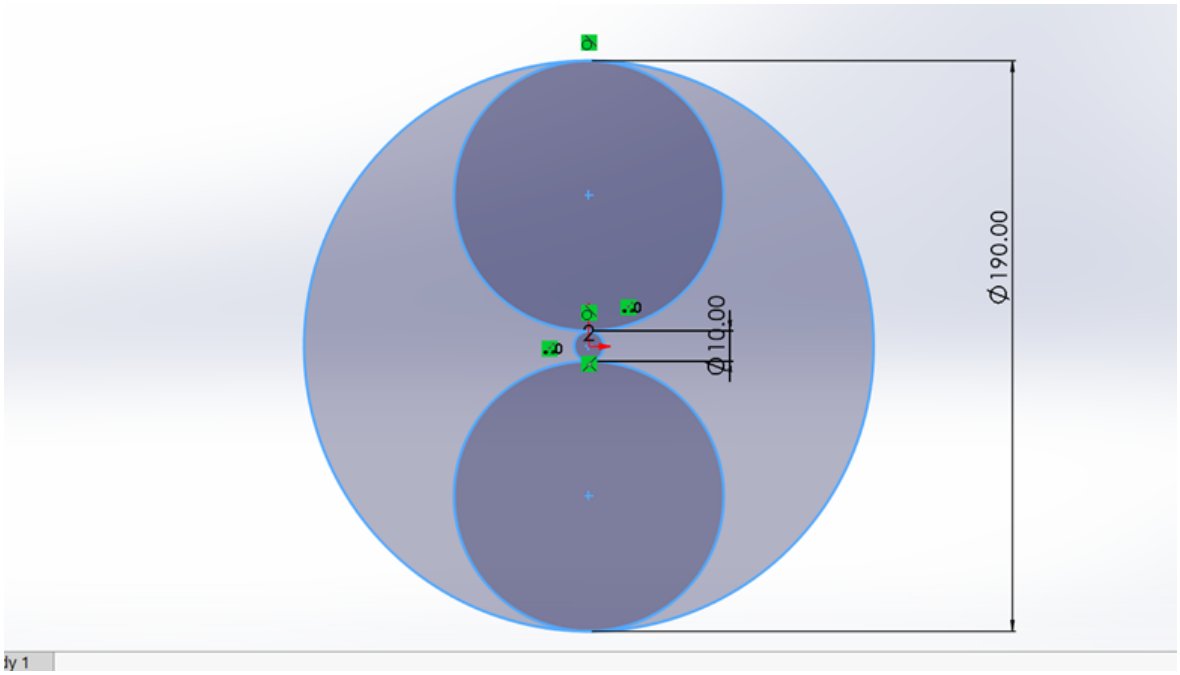


Figure 3.1: Add caption here

3.5 3D MODELING

Following components were modelled using SolidWorks:

3.5.1 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.

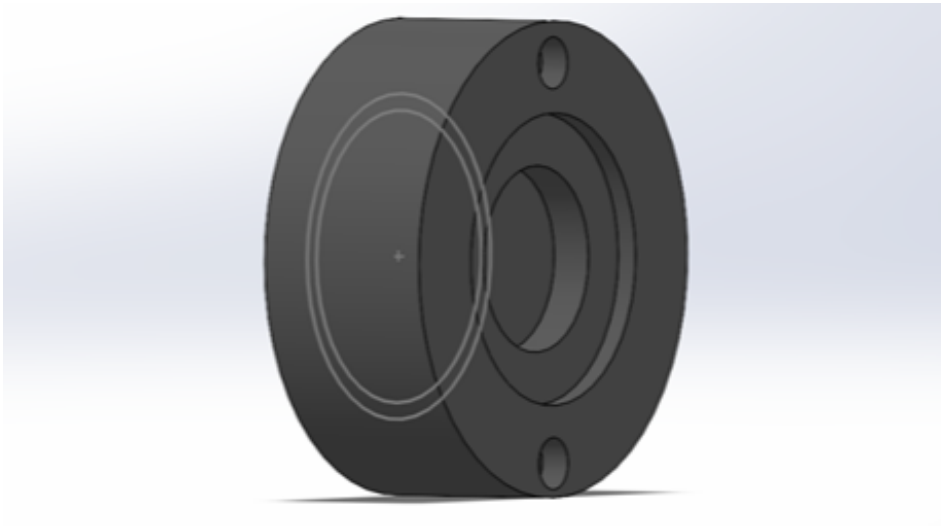


Figure 3.2: Caption 1

3.5.2 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.

3.5.3 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.

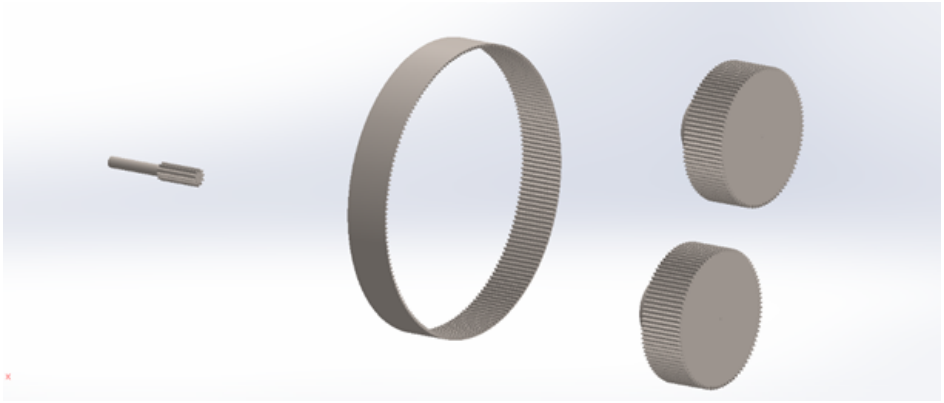


Figure 3.3: Caption 1

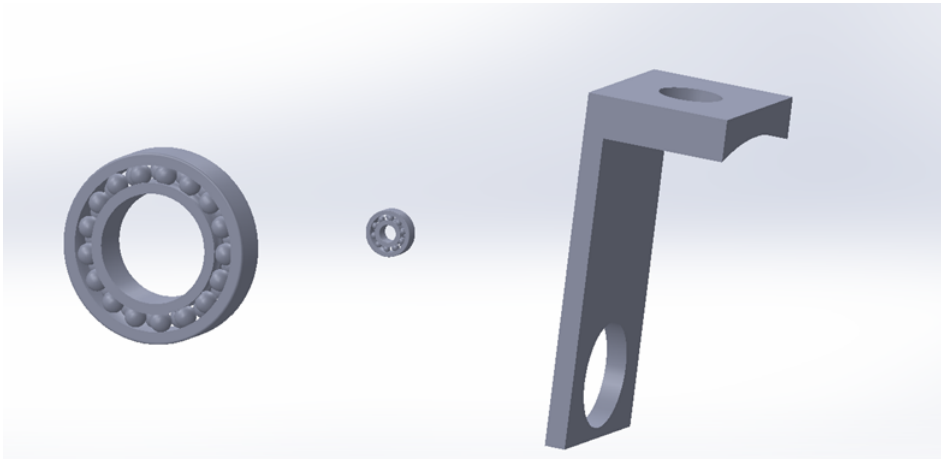


Figure 3.4: Caption 1

3.5.4 Housing:

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

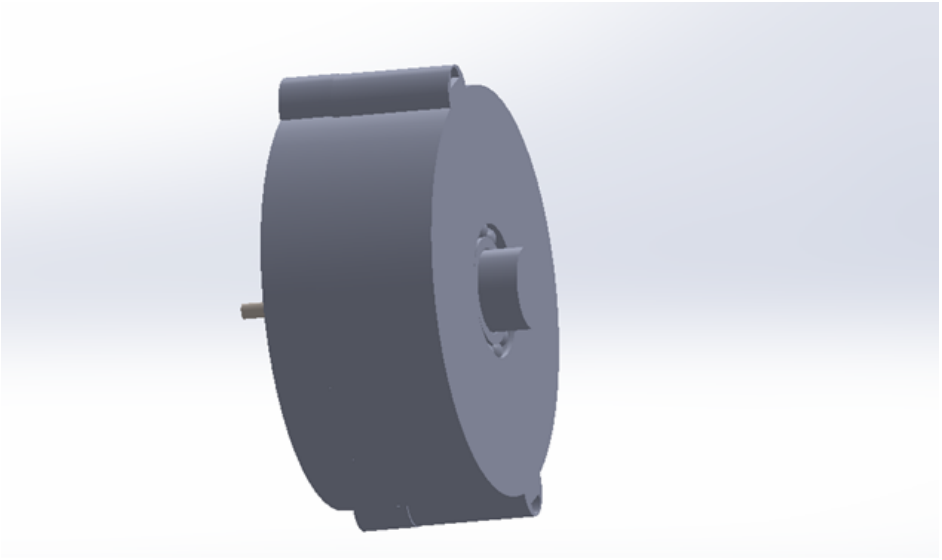


Figure 3.5: Caption 1

3.5.5 Assembly

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

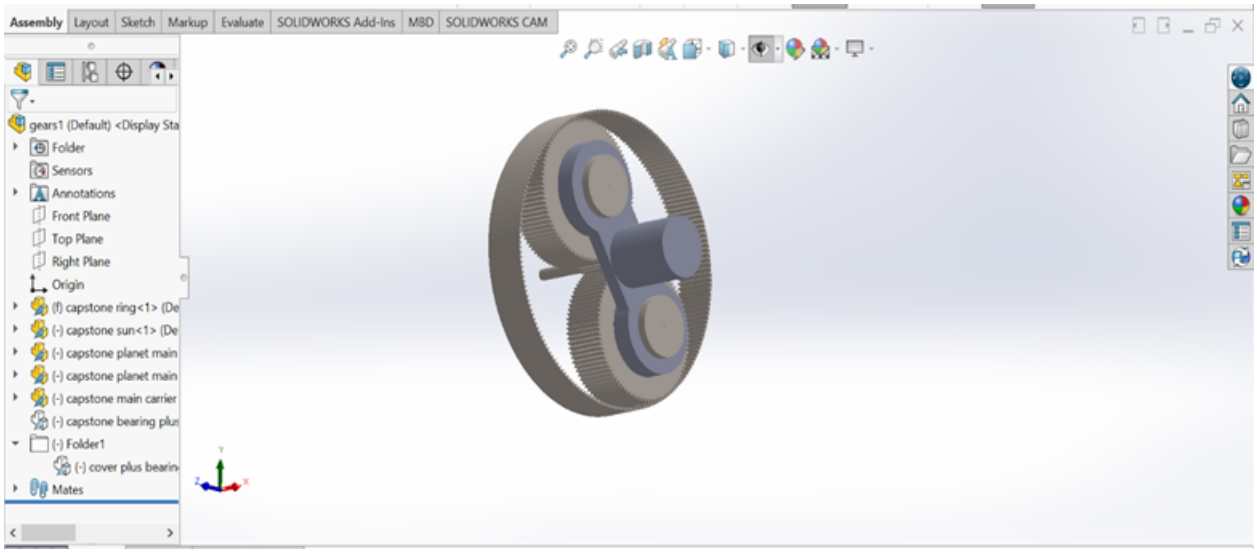


Figure 3.6: Add caption here

3.6 CALCULATION AND ANALYSIS

Simulations done in SolidWorks to test performances of the design:

3.6.1 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]

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

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

3.7 MOTION ANALYSIS:

Wheels' rotation and torque transfer in the gearbox simulated.

Result: Smooth motion with efficiency in power transmission.

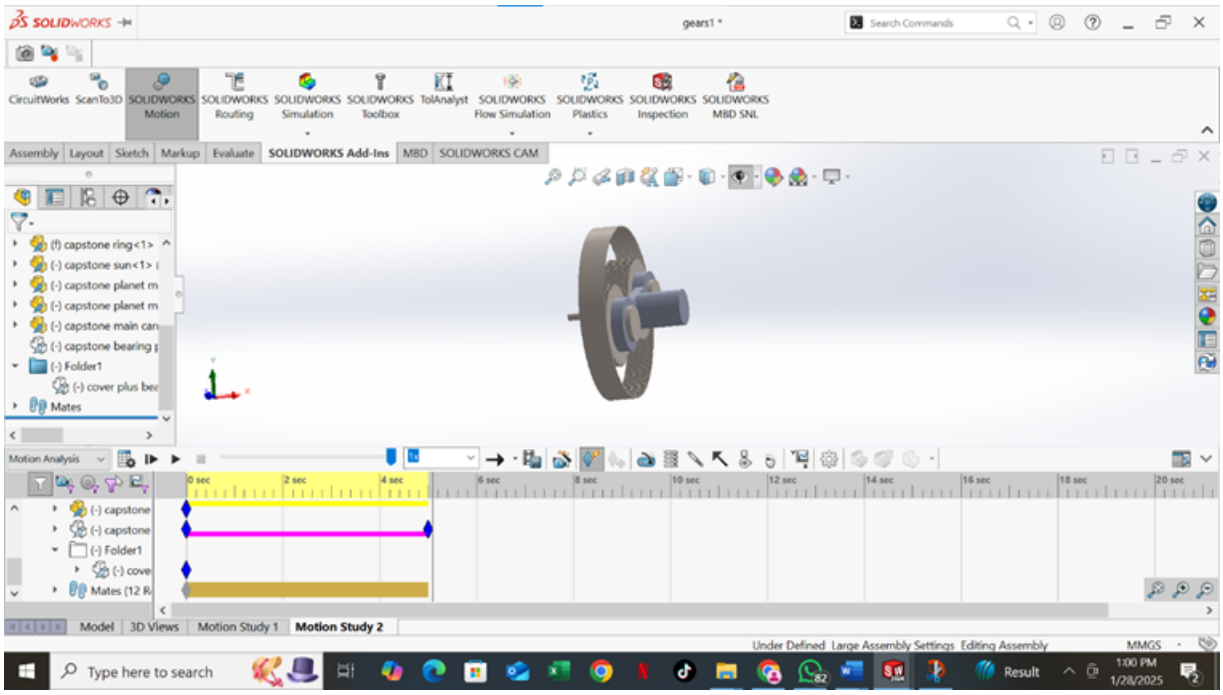


Figure 3.7: Add caption here

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

Results and Discussion

The final design was able to meet all the specified requirements, which included load capacity, torque output, and manufacturability. The simulations proved that the system was structurally sound and could operate without hitches under realistic conditions .

3.8 CONCLUSION

It should not only be functional, but also practical, manufacturable, and economic, taking into consideration all the constraints. The literature survey focuses on the fact that the gearbox-equipped drive wheel system is a main concern in the design of an AGV. Material selection, load-carrying capacity, torque optimization, and manufacturability are the key factors to be considered. In this paper, design and simulation for performance improvement of the AGV using SolidWorks are proposed to overcome these bottlenecks and help in the development of reliable high-performance drive systems for AGVs.

The middle drive wheels with an integrated gearbox were designed and further validated for use in an AGV. The system was reliable, efficient, and easy to manufacture. Future work can be the physical development of the prototype and performance testing at site applications..

Chapter 4

gregory's chapter

4.1 INTRODUCTION

Bearing wheels and caster wheels are integral components in various industrial and commercial applications, facilitating mobility, load distribution, and operational efficiency. These wheels are commonly found in material handling equipment, automotive systems, and heavy machinery, where they ensure seamless movement and reduced friction. The efficiency and durability of these components directly impact productivity, safety, and cost-effectiveness in different industries.

Designing an effective bearing or caster wheel system involves several critical factors, including material selection, load-bearing capacity, resistance to environmental conditions, and compliance with industry standards such as ISO 3691-4:2020. Material properties play a vital role in determining the strength, wear resistance, and overall performance of the wheels. Common materials used in bearing wheels include AISI 1045 steel for shafts and AISI 52100 chrome steel for bearings, both of which offer high durability and load-handling capabilities.

The study of bearing and caster wheels also involves analyzing mechanical constraints, including torque requirements, frictional forces, and stress distribution. Ensuring optimal performance requires comprehensive calculations, including shaft diameter determination, bearing life assessment, and stress-strain analysis. Additionally, environmental considerations such as exposure to moisture, temperature variations, and corrosive elements influence the selection of materials and protective coatings.

Manufacturing constraints must also be addressed to maintain cost-effectiveness and efficiency in production. Precision machining, heat treatment processes, and surface finishing techniques are necessary to achieve the desired durability and performance characteristics. Furthermore, safety factors such as load limitations, stability, and braking mechanisms must be considered to prevent operational hazards and equipment failure.

This project aims to develop a systematic approach to designing and evaluating bearing

and caster wheels while addressing real-world challenges. By integrating theoretical calculations with finite element analysis (FEA), this study seeks to enhance the reliability, efficiency, and sustainability of these components. The findings will contribute to improving material selection, optimizing load distribution, and ensuring compliance with industry regulations, ultimately leading to more robust and long-lasting wheel systems.

4.2 LITERATURE REVIEW

Research on bearing wheels and caster wheels has extensively explored their mechanical properties, material composition, and industrial applications. Several studies have emphasized the significance of material selection in determining the durability and efficiency of these components. According to Smith et al. (2020), AISI 1045 steel is widely used for shaft manufacturing due to its high tensile strength and resistance to mechanical fatigue. Similarly, AISI 52100 chrome steel is preferred for bearings due to its excellent hardness, wear resistance, and ability to withstand high loads.

Lubrication and friction management play crucial roles in ensuring the longevity of bearing systems. Studies by Brown and Pietra (2018) highlight that improper lubrication can lead to increased friction, wear, and premature failure of bearings. Advanced lubrication techniques, such as sealed bearings and self-lubricating materials, have been explored to enhance performance and minimize maintenance requirements.

The impact of environmental factors on bearing and caster wheel performance has also been a key area of research. According to Can and Ozkarahan (2019), exposure to high humidity and extreme temperatures can accelerate corrosion and degrade material integrity. Protective coatings, such as zinc plating and polymer encapsulation, have been recommended to improve resistance against environmental hazards.

Manufacturing techniques and precision engineering have further influenced the development of high-performance bearing and caster wheels. Recent advancements in computer-aided design (CAD) and finite element analysis (FEA) have enabled engineers to optimize designs for maximum efficiency. Studies by Johnson et al. (2021) indicate that simulation-based testing allows for accurate prediction of load distribution, stress concentrations, and potential failure points, leading to more reliable designs.

Safety considerations in wheel design have also been extensively analyzed. Research by Lee and Kim (2022) emphasizes the importance of braking mechanisms and stability enhancements in caster wheel applications, particularly in medical and industrial equipment. Proper weight distribution and impact resistance are essential factors in preventing accidents and improving operational safety.

In conclusion, the literature underscores the importance of material selection, lubrication,

environmental protection, and advanced manufacturing techniques in optimizing bearing and caster wheel performance. The integration of modern engineering tools and industry standards ensures that these components meet the growing demands of industrial applications while maintaining safety, durability, and efficiency. This study builds upon existing research to develop a comprehensive approach to designing and evaluating bearing and caster wheel systems.

4.3 REALISTIC CONSTRAINTS

4.3.1 Material Constraints

Bearing wheels and caster wheels require high-strength materials for load-bearing capacity and durability. AISI 1045 steel is chosen for the shaft due to its excellent mechanical properties, including high yield strength (530 MPa) and good machinability. AISI 52100 chrome steel is used for bearings due to its superior hardness and wear resistance. However, these materials are susceptible to corrosion in humid environments, necessitating protective coatings or stainless alternatives. The choice of rubber or polyurethane for the wheel itself impacts rolling resistance, wear rate, and temperature resistance.

4.3.2 Manufacturing Constraints

Manufacturing challenges include precision machining of shafts and bearings to ensure proper tolerances. Heat treatment processes are required for AISI 52100 bearings to achieve the necessary hardness. Forging and casting limitations impact cost and production scalability. Bearings must meet ISO 3691-4:2020 standards, which impose strict requirements on tolerances and material integrity. The availability of raw materials and manufacturing costs also influence design decisions.

4.3.3 Mechanical Constraints

Bearing wheels and caster wheels must withstand dynamic loads and impacts without failure. Bearings must be designed to handle radial and axial loads efficiently. Torque calculations ensure that the system operates within safe stress limits. Misalignment and excessive vibration can reduce bearing life, requiring proper lubrication and mounting techniques. The wheel's rolling friction, influenced by surface conditions and load distribution, impacts efficiency.

4.3.4 Environmental Constraints

Environmental factors such as temperature variations, moisture, and chemical exposure affect material performance. Bearings exposed to extreme temperatures may experience reduced lubrication effectiveness, leading to higher friction and wear. Caster wheels used in outdoor environments must resist UV degradation and corrosion. Sustainable material choices and waste reduction strategies are essential to meet environmental regulations.

4.3.5 Safety Constraints

Safety considerations include ensuring that the wheels can withstand maximum load without failure. Overloading can lead to bearing fatigue and catastrophic failure. Proper braking mechanisms must be integrated into caster wheel systems to prevent uncontrolled movement. Bearings must comply with safety standards such as ISO 3691-4:2020 to prevent accidents due to bearing failure. Ergonomic considerations also play a role in reducing strain on operators handling heavy loads.

4.4 CHAPTER FOUR: METHODS USED TO DESIGN THE PROJECT

4.4.1 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.

4.4.2 Methodology

4.4.3 Bearing selection

4.4.3.1 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.

4.4.3.2 Shaft Diameter Calculation

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

$$\omega = \frac{v}{R} \tag{4.1}$$

Substitute values in eq. (4.1):

$$\omega = \frac{1.25}{0.110} = 11.36 \text{ rad/s}$$

Convert ω to RPM:

$$RPM = \omega \times 602\pi = 11.36 \times 602\pi \approx 21484.99 \text{ RPM}$$

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:

$$M = F \times L \tag{4.2}$$

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:

$$d = (16M/\pi\tau max)^{1/3} \tag{4.3}$$

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. (4.3):
$d = (16 \times 49.05/\pi \times 62.5 \times 106)^{1/3}$= 0.012mm

4.4.3.3 Bearing Selection

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

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

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.

$C = 981 \times (10^6/1,000,000)^{1/3} = 981N$

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 :**

4.4.3.4 Material Properties of AISI 1045 Steel

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

4.4.3.5 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.

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

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:

$$\text{z} = \frac{\pi(D+d)}{2d_b} \tag{4.4}$$

where:

- z is the number of balls.
- D is the outer diameter of the bearing.
- d is the inner diameter of the bearing.
- db is the diameter of each ball.

Given:

- z=9
- D = 32 mm
- d = 12 mm

We need to solve for db:
$9=\pi(32+12)/2db$
$9=\pi\times44/2db$
$db=22\pi/9$
$db\approx7.68$ mm
For a 6201 bearing:

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

4.4.3.6 Material Selection

Shaft Material: AISI 1045 Steel

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

Bearing Material: AISI 52100 Chrome Steel

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

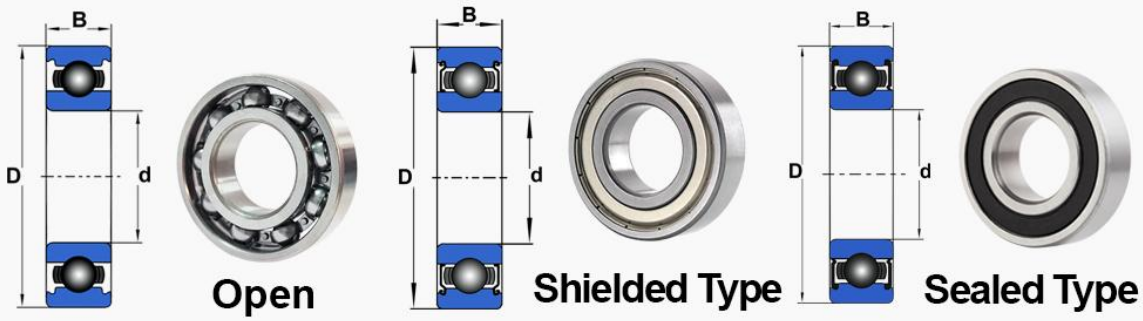
4.4.3.7 Compatibility Check

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)		Basic Load Ratings {kgf}		Factor	Limiting Speeds (rpm) Grease		Limiting Speeds (rpm) 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 4.1: caption here not added by Gregory

4.4.4 Torque calculation

4.4.4.1 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)

4.4.4.2 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.

4.4.5 Results

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

4.4.5.1 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.

4.4.5.2 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.

4.4.5.3 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.

4.4.5.4 Total Torque Required:

The dynamic load needed is 28.2Nm

4.5 CONCLUSION AND FUTURE WORKS

The selection and design of bearing wheels and castor wheels require careful consideration of material properties, mechanical performance, and environmental factors to ensure durability and efficiency. In this project, the NSK 6201 bearing was chosen for its high performance and reliability. The NSK 6201 bearing is a single-row deep groove ball bearing known for its ability to handle both radial and axial loads, making it ideal for various industrial applications.

SolidWorks was used for 3D modeling and simulation, providing an accurate representation of the bearing and castor wheel assemblies. The finite element analysis (FEA) conducted within the software helped identify potential failure points and refine the design to enhance structural integrity. This approach significantly improved the efficiency of the design process, reducing the need for physical prototyping and minimizing costs.

The design and manufacturing process adhered to the ISO 3691-4:2020 guidelines, which specify safety requirements and verification for driverless industrial trucks and their systems. These guidelines ensure that the bearing and castor wheels meet the necessary safety standards, including braking systems, speed control, load handling, and stability. Compliance with these standards is crucial for the safe and efficient operation of industrial trucks.

Future work should focus on real-world validation of the proposed design through experimental testing and field applications. Additionally, the integration of smart sensors for real-time load monitoring and predictive maintenance could enhance operational efficiency and extend the service life of the components. Sustainable material alternatives should also be explored to align with environmental regulations and industry trends toward eco-friendly solutions.

In conclusion, this project successfully demonstrated a structured approach to the design and analysis of bearing wheels and castor wheels. By leveraging SolidWorks for design validation and incorporating theoretical and empirical analysis, the study provides a comprehensive framework for developing durable and high-performance wheel systems. Further

advancements in materials and digital manufacturing technologies will continue to improve these components, ensuring reliability in industrial applications.

Chapter 5

ROS

we will introduce basic ROS concepts and the ROS package management system in order to approach ROS programming. In this chapter, we will go through ROS concepts such as the ROS master, the ROS nodes, the ROS parameter server, and ROS messages and services, all while discussing what we need to install ROS and how to get started with the ROS master. In this chapter, we will cover the following topics:

- Why should we learn ROS?
- Understanding the ROS filesystem level.
- Understanding 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 5.1: Ubuntu 20.04

5.1 WHY SHOULD WE USE 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:

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

5.2 UNDERSTANDING 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:

- **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

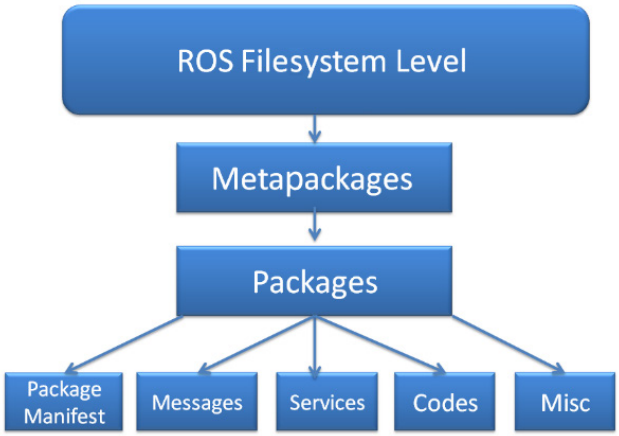


Figure 5.2: ROS filesystem level

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 5.3: List of files inside the package

5.3 ROS PACKAGES

The typical structure of a ROS package is shown here:

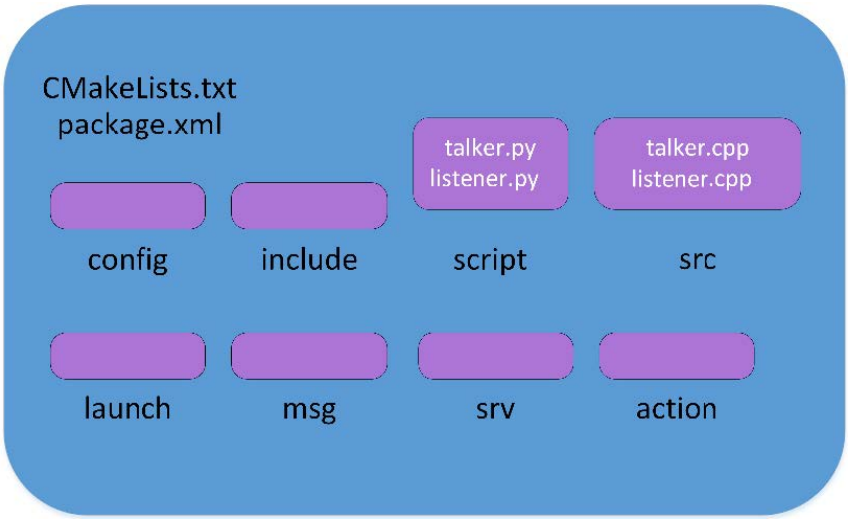


Figure 5.4: Structure of a typical C++ ROS package

- **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. We will learn more about these kinds of files in the next chapter.
- **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: 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.

```
<?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>
```

Figure 5.5: Structure of package.xml

5.4 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:

```
<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:

```
roscd navigation
```

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

```
gedit package.xml
```

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

```
<?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>
```

Figure 5.6: Structure of the package.xml metapackage

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

5.5 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:

```
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 5.1: 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:

```
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.

5.6 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:

```
#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.

5.7 UNDERSTANDING 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**.

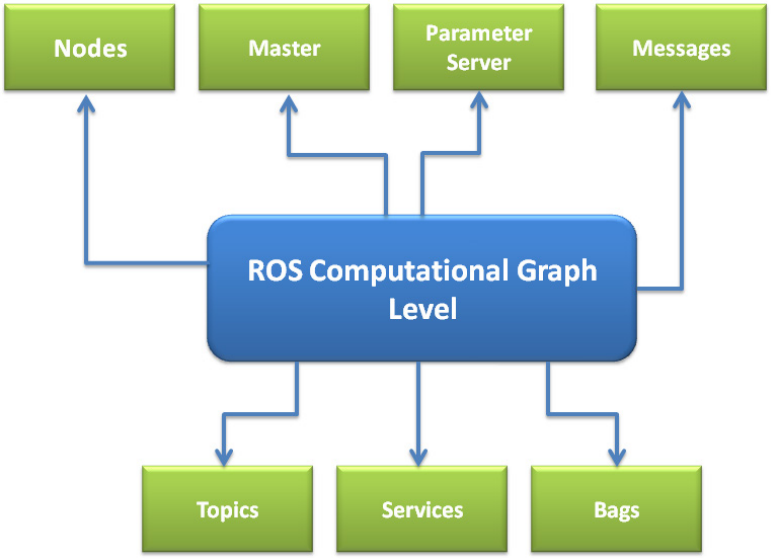


Figure 5.7: Structure of the ROS graph layer

5.7.1 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.

5.7.2 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.

5.7.3 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.

5.7.4 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.

5.7.5 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:

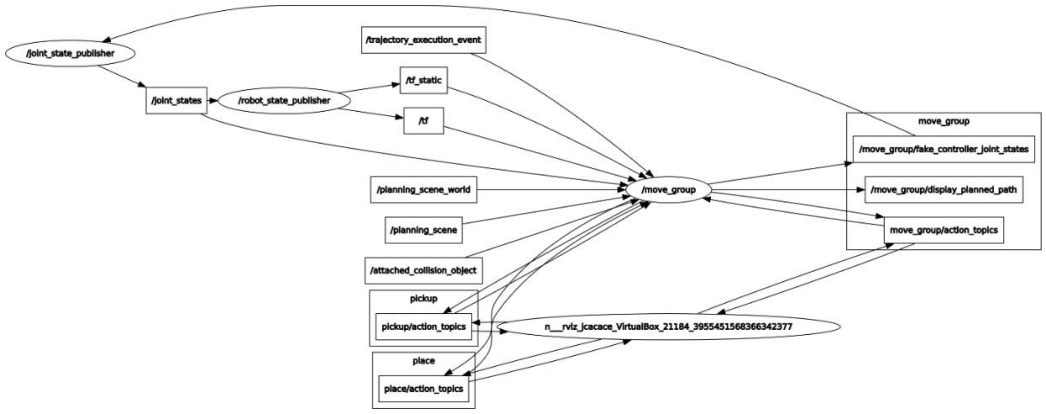


Figure 5.8: Graph of communication between nodes using topics

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).

5.8 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.

5.9 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.

5.10 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.

5.11 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.

5.12 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:

```
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`.

5.13 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 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

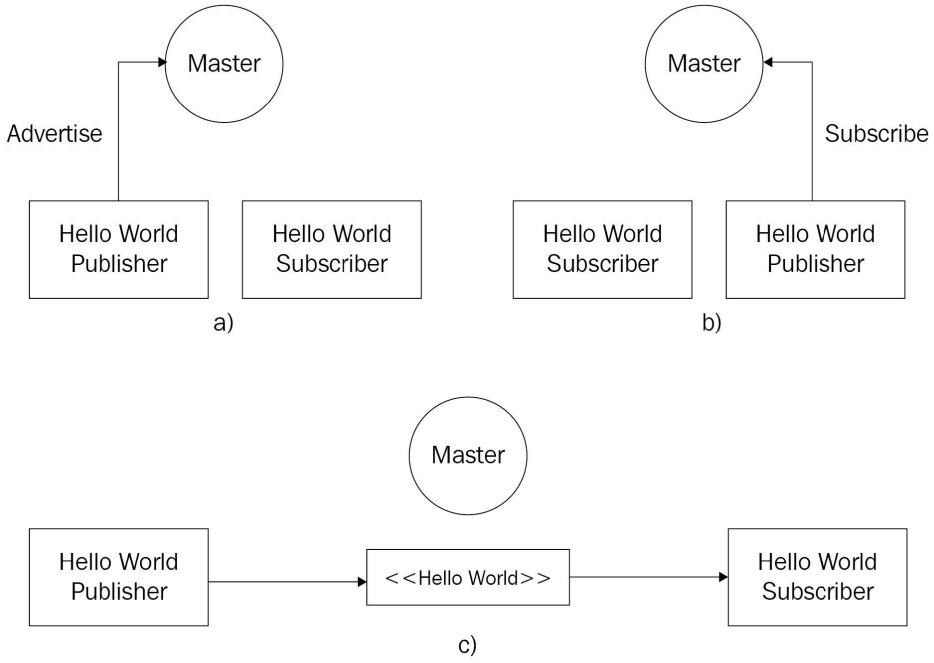


Figure 5.9: Communication between the ROS master and Hello World publisher and subscriber

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.

5.14 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`).

5.15 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 5.3: ROS Distributions Table

5.16 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:

```
roscore
```

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

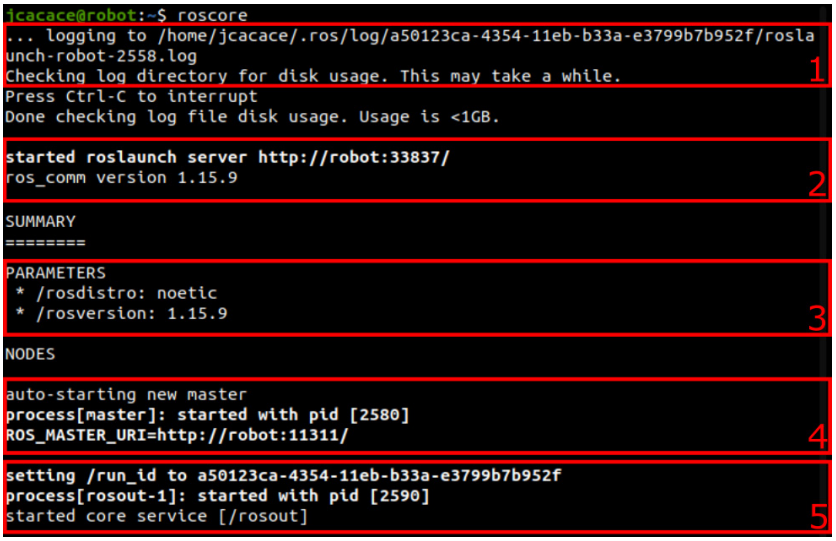


Figure 5.10: Terminal messages while running the roscore command

- 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`:

```
<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.

5.16.1 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:

```
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:

```
/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:

```
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:

```
/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:

```
rosservice list
```

The list of services that are running is as follows:

```
/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.

Chapter 6

SIMULATION ENVIRONMENT

6.1 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.

6.2 TOOLS AND FRAMEWORK

6.2.1 Gazebo

Gazebo was initially developed in 2002 to facilitate the simulation of ground robot applications in both indoor and outdoor environments [6]. 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 6.1: 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) [7], Bullet [8], Simbody [9], and the Dynamic Animation and Robotics Toolkit (DART) [10]. These engines handle rigid-body physics simulations, providing highly accurate interactions between objects. Additionally, Gazebo employs the Object-Oriented Graphics Rendering Engine (OGRE) [11] 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.

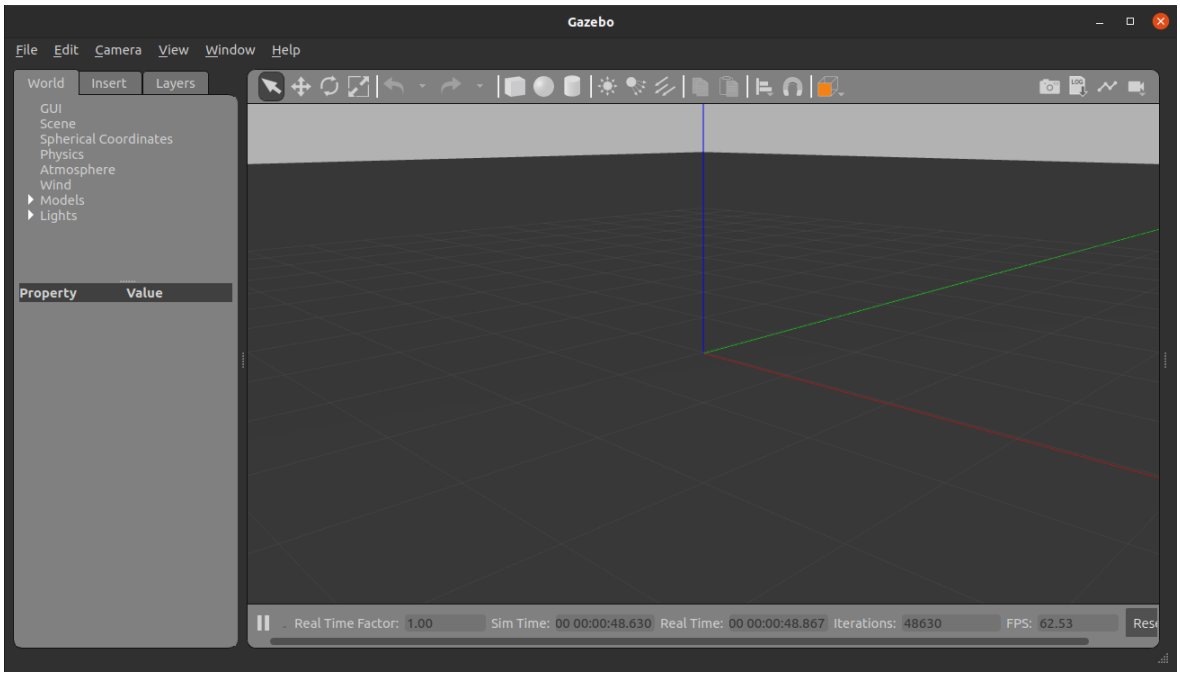


Figure 6.2: Empty world in Gazebo

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.

```
  <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. 6.3 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.

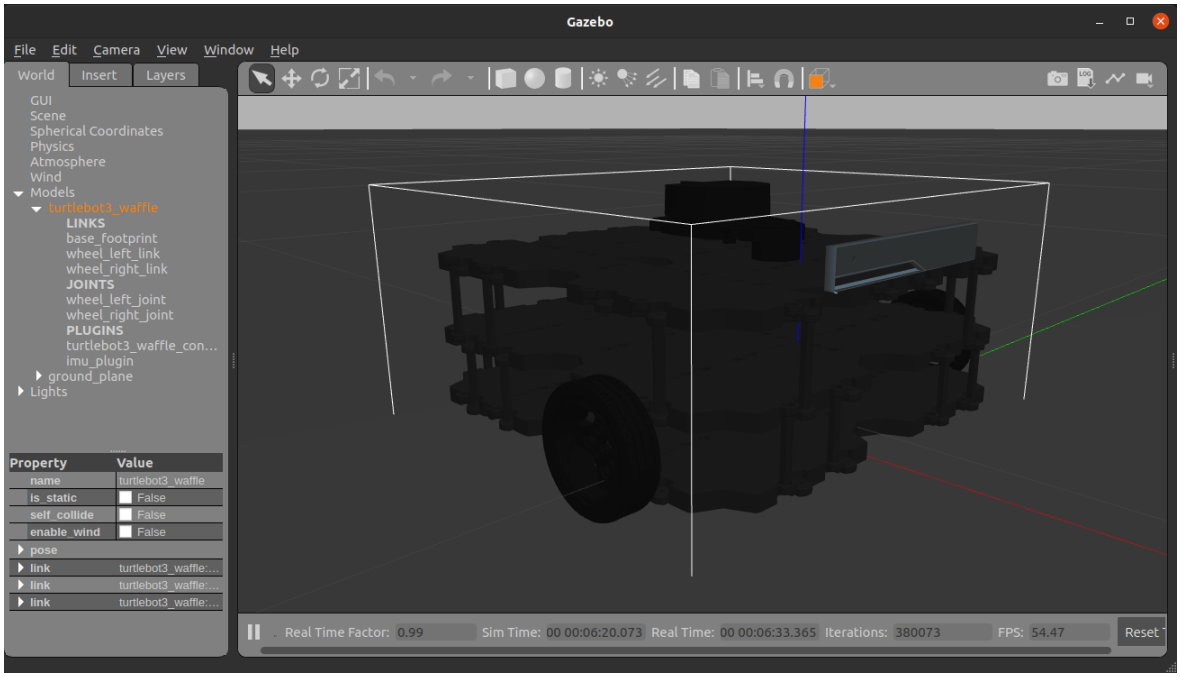


Figure 6.3: TurtleBot3 Waffle spawned in empty world

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. 6.4 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.

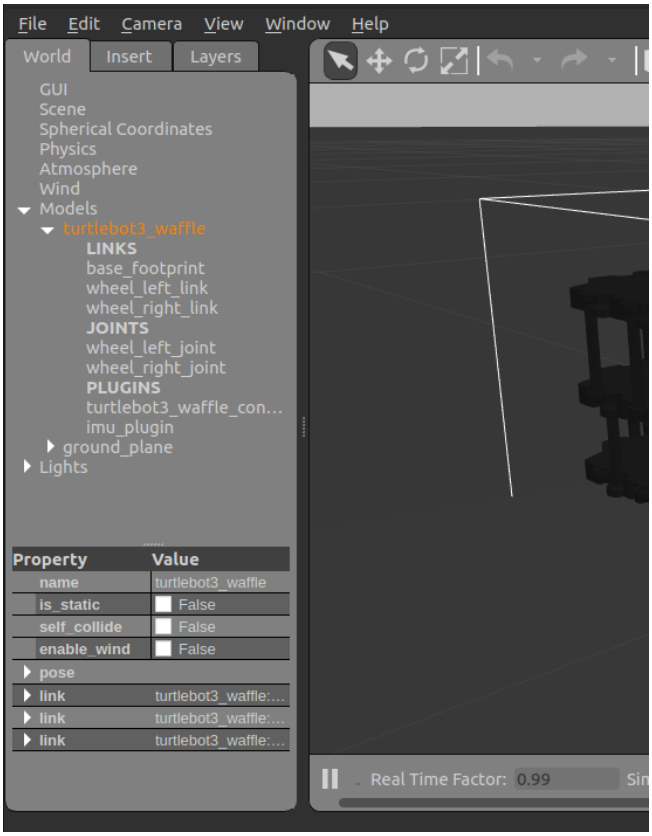


Figure 6.4: Gazebo Graphical User Interface left side pannel

The lower part of the Gazebo GUI displayed in fig. 6.5 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.

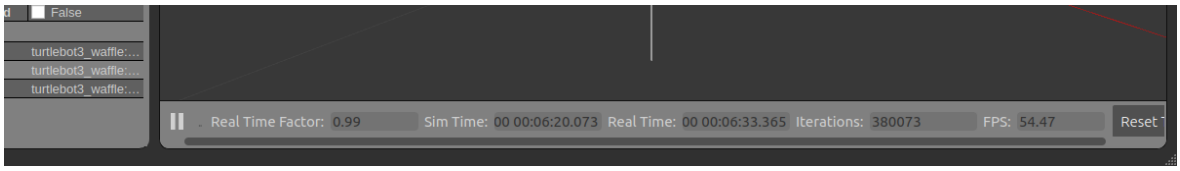


Figure 6.5: Gazebo Graphical User Interface lower part

The top part of the Gazebo GUI (fig. 6.6) 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.

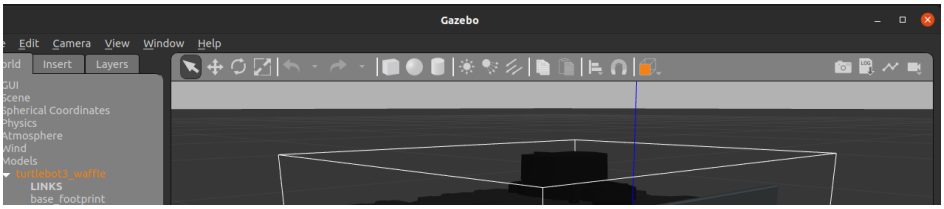


Figure 6.6: Gazebo Graphical User Interface top part

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.

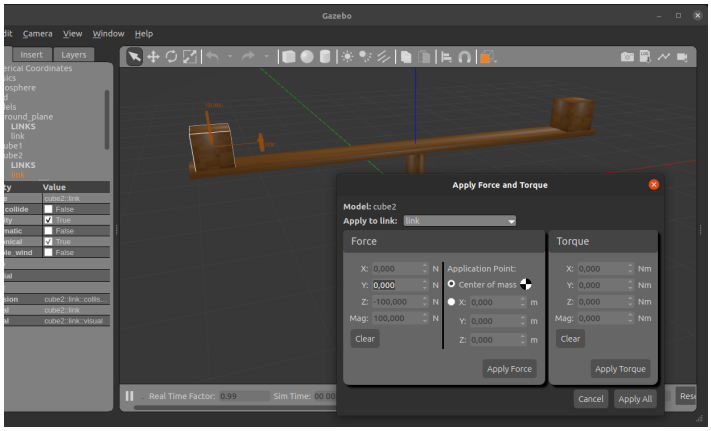


Figure 6.7: Example of force applied on one side of a seesaw in gazebo

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. 6.7 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.

6.2.2 RViz

RViz (ROS Visualization) is an open-source 3D visualization tool that provides real-time graphical representations of robotic systems.

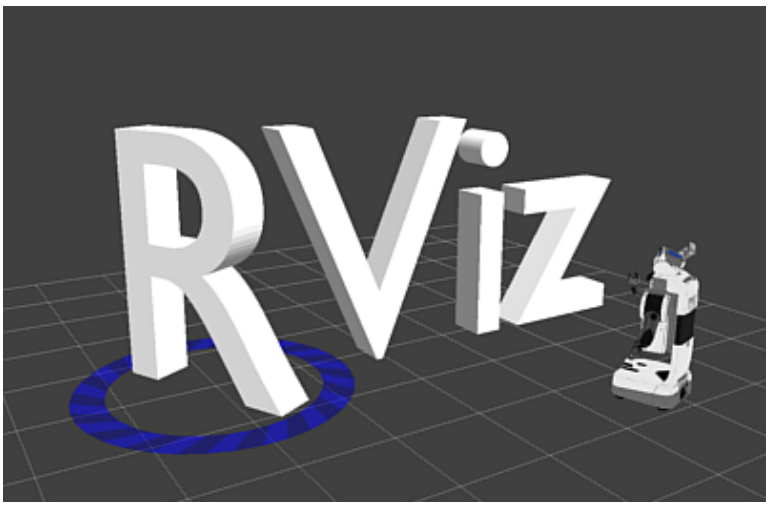


Figure 6.8: Rviz (Ros Visualization)

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.

6.3 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 [12].

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.

6.3.1 Robot Model (URDF/SDF)

6.3.2 World Design

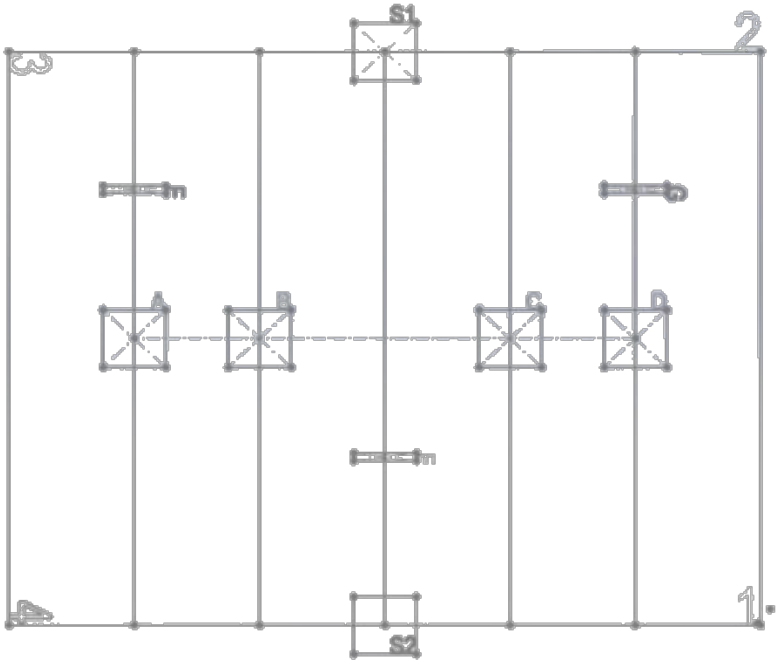


Figure 6.9: Teknofest competition real map layout

The simulation environment was designed to replicate the competition layout in fig. 6.9 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.

6.3.2.1 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:

- **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.

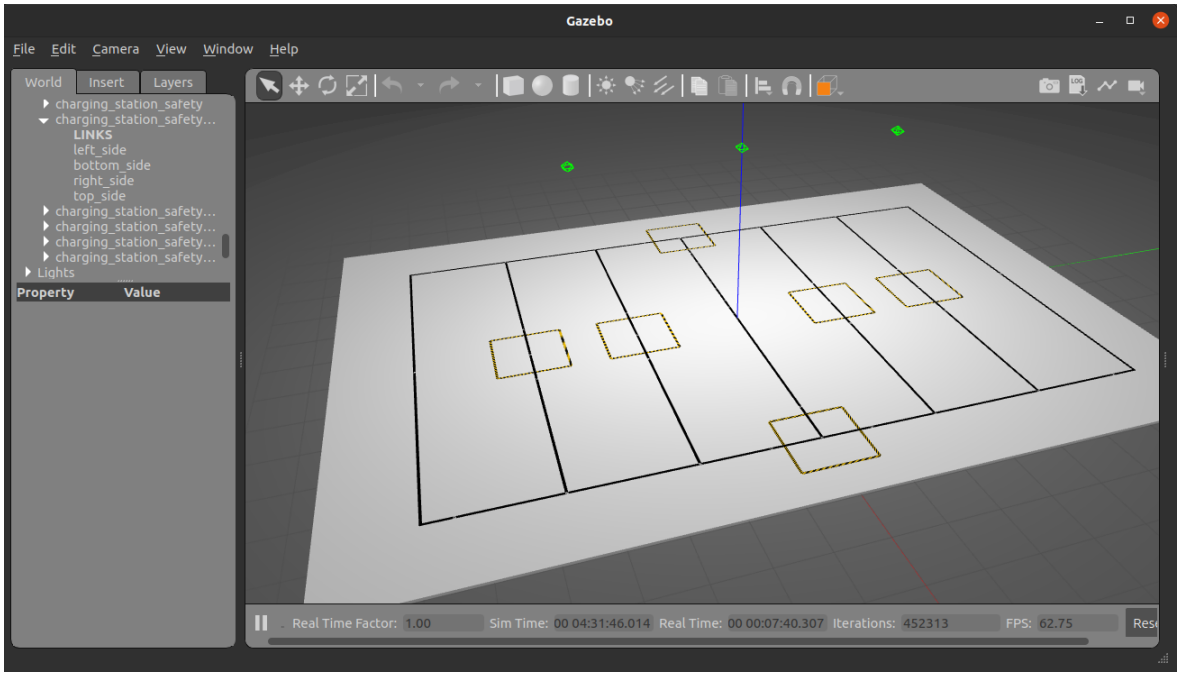


Figure 6.10: White wooden floor with black lines in gazebo

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.

```
...
<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>
  ...
```

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.

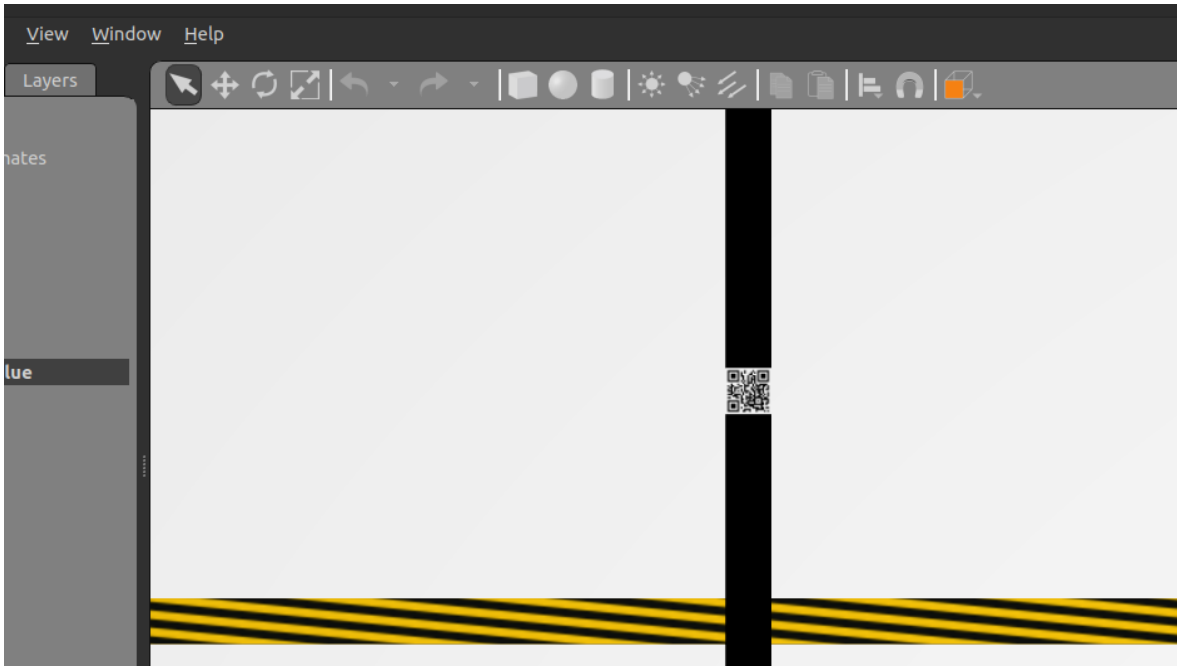


Figure 6.11: Example of Qr Code tag before a loading point

the `<friction>` tag parameters of the white floor, displayed in fig. 6.10, 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.

```
<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>
```

- **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.

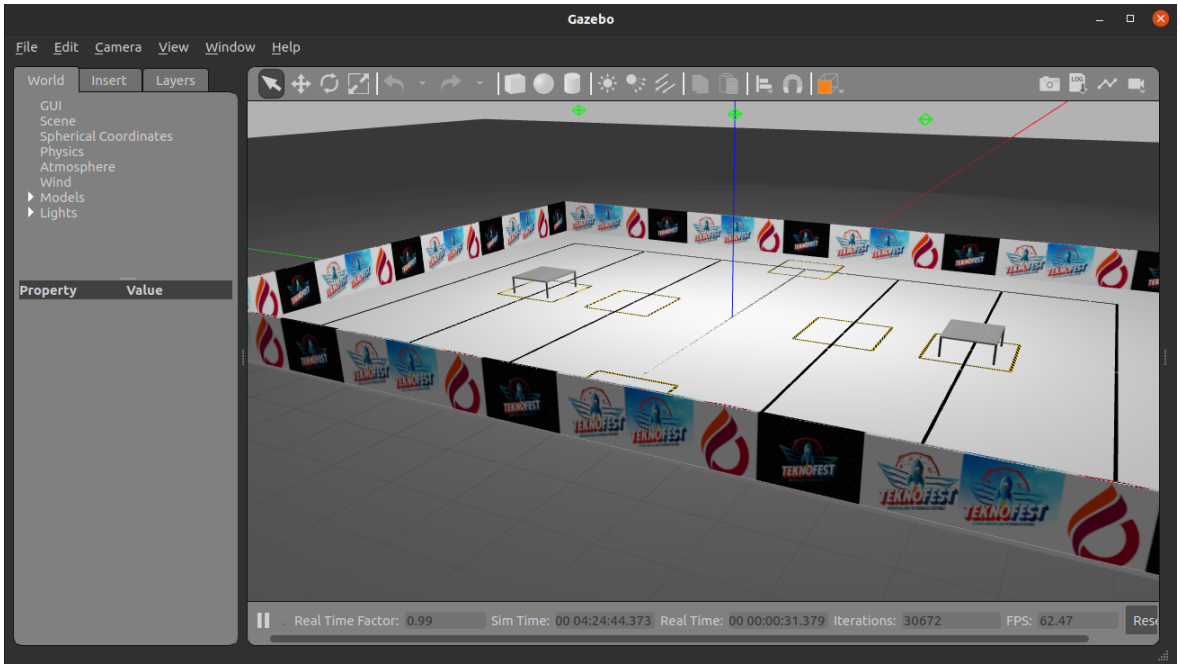


Figure 6.12: Competition aera bounded by ad hoardings

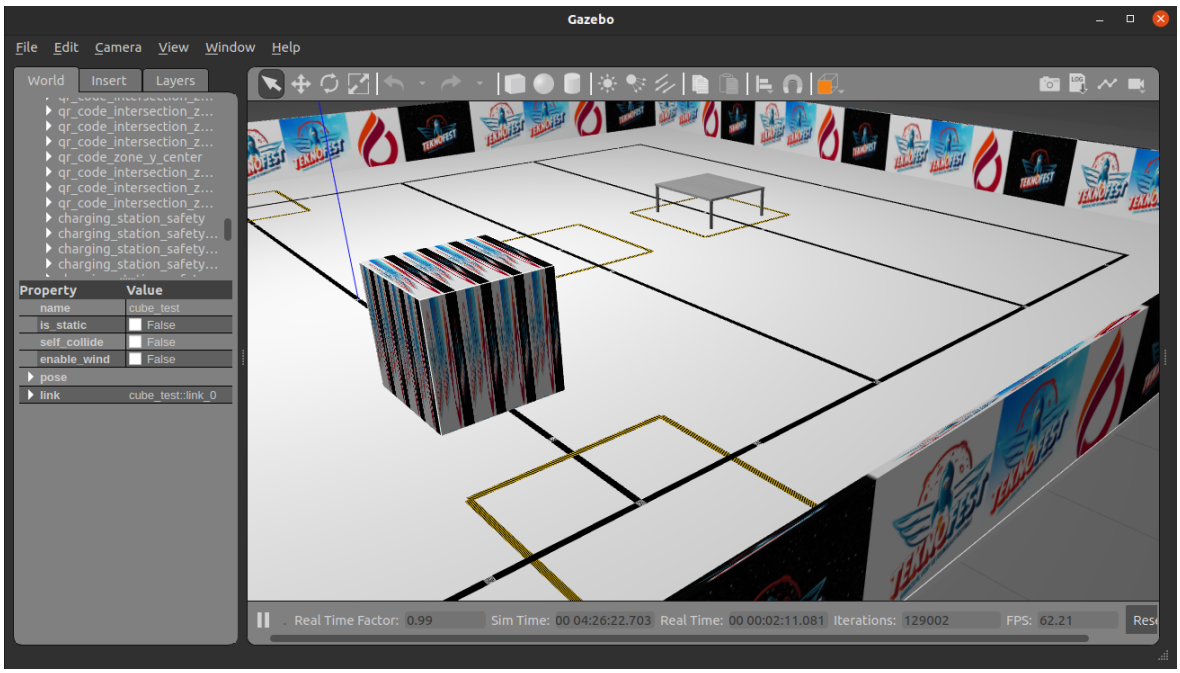


Figure 6.13: Competition area line interrupted by permanent obstacle

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.

```
<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>
    ...
```

```
      ...
      <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>
      ...
```

```
    ...
    <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.

```
...
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_contact_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_impulse_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.

```
<?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 above SDF code 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. 6.9, 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.

```
    ...
    {
      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 6.14: Qr code tag placed at the intersection of line of the zone C and the station 2

- **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.
- **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. 6.15 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.

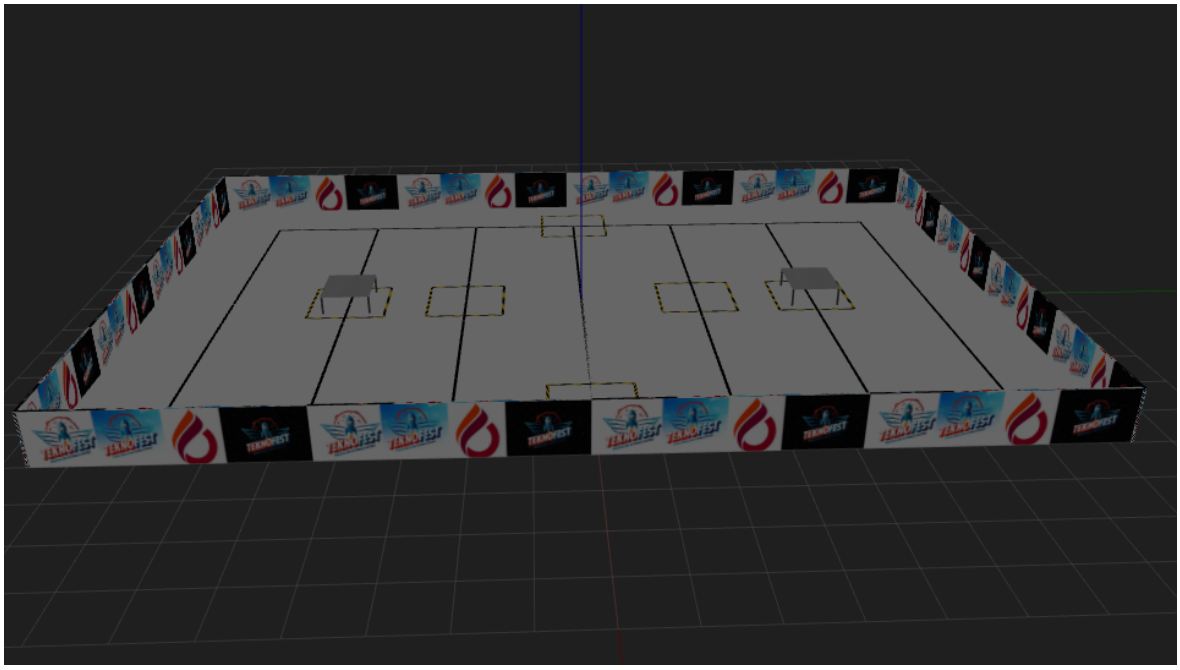


Figure 6.15: Effect of lighting conditions in simulation environment

On the other hand, good lighting shown in fig. 6.16 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.

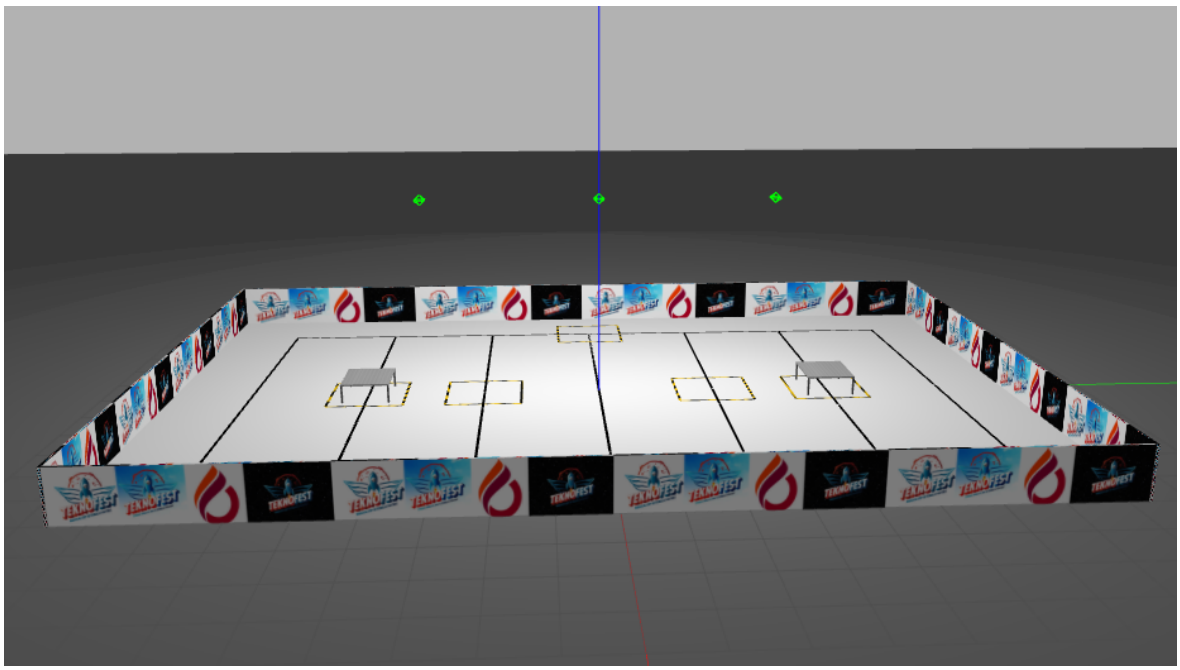


Figure 6.16: Effect of lighting conditions in simulation environment

6.3.2.2 Physics Configuration

I should maybe say more about this

6.3.2.3 Realism Enhancements

Is it necessary ?

6.3.3 Control and sensor Integration

You should be able to do something about this

6.4 VALIDATION OF THE SIMULATION

6.5 CHALLENGES AND SOLUTIONS

Part III

METHODS USED TO DESIGN THE PROJECT

Chapter 7

URDF

7.1 UNDERSTANDING 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:

7.1.1 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:

```
<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:

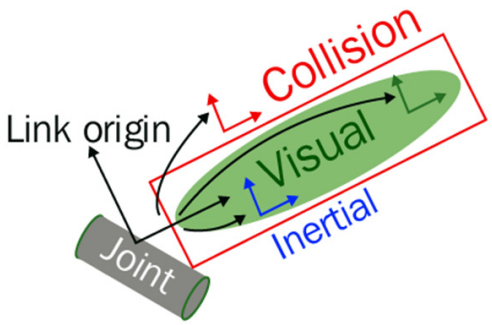


Figure 7.1: A visualization of the URDF link [1]

7.1.2 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:

```
<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:

7.1.3 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:

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

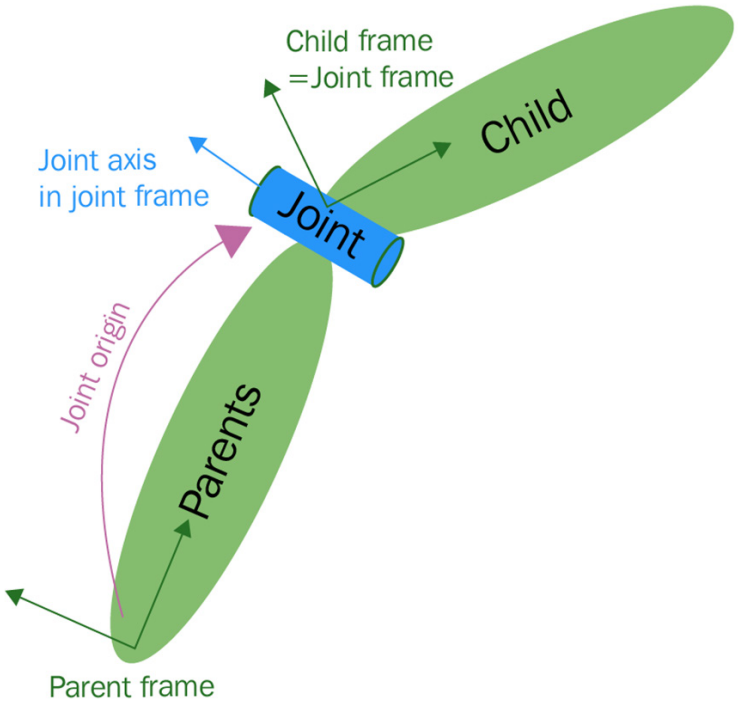


Figure 7.2: A visualization of the URDF joint [1]

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

7.1.4 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:

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

You can find more about URDF tags at wiki.ros [13]. 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.

7.1.5 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

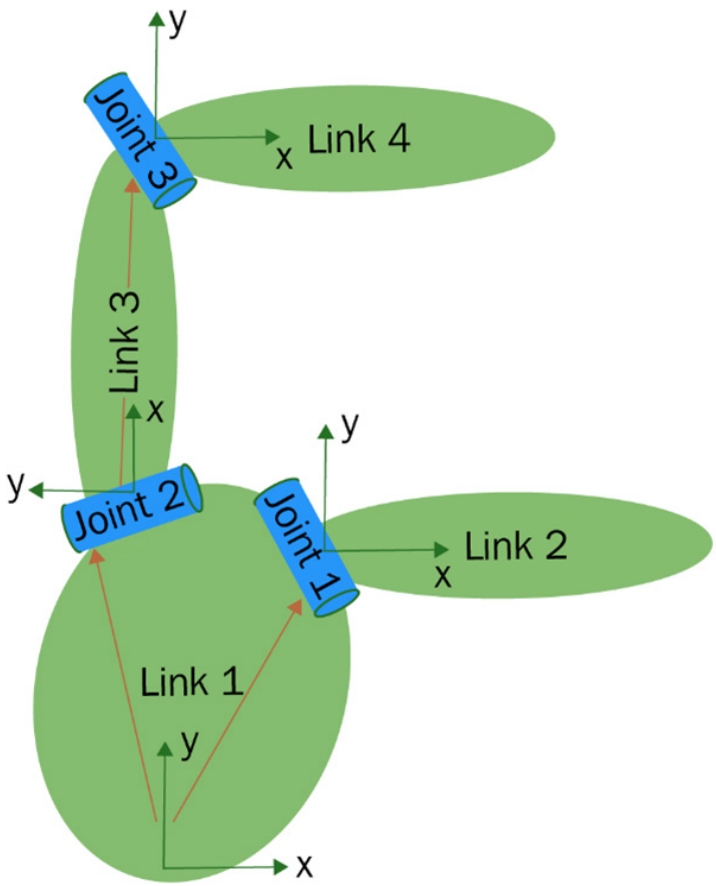


Figure 7.3: A visualization of a robot model with joints and links [1]

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:

```
    <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>
```

7.1.6 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:

```
<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">
    <mechanicalReduction>100.0</mechanicalReduction> <!-- Gear ratio -->
  </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 summery:

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 7.1: Essential Dependencies for Simulating a URDF in Gazebo

7.2 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.

7.2.1 Understanding 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:

```
<?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.

7.2.2 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:

```
<!-- 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:

```
<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, the value 0.9, is replaced with ”base_length”, and ”0.85” is replaced with ”base_width”.

7.2.3 Using the math expression

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:

```
<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>
```

7.2.4 Using 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:

```
<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, 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:

```
<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"/>
```

7.2.5 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:

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

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

7.3 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:

```
<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>
```

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:

```
roslaunch the_pkg_name launch_file_name.launch
```

If everything works correctly, we will get a pan-and-tilt mechanism in RViz, as shown here:

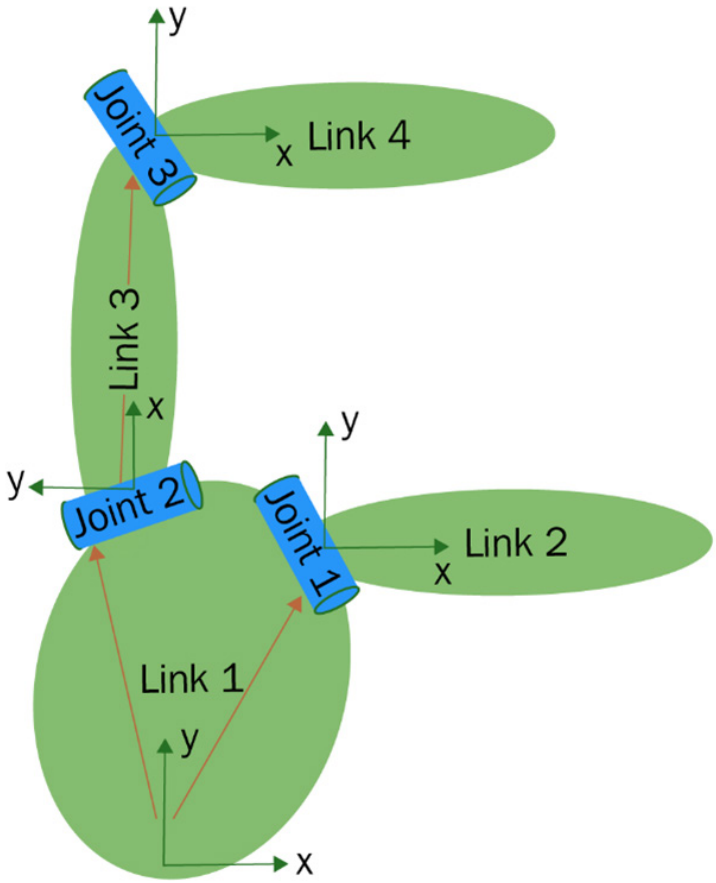


Figure 7.4: A visualization of a robot model with Rviz

7.3.1 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:

```
<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:

```
<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>
```

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: In this user

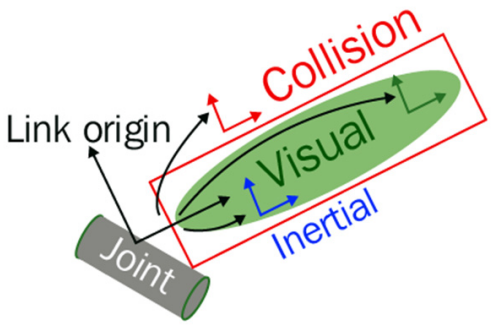


Figure 7.5: The joint level of the platform lifting mechanism

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.

Chapter 8

NAVIGATION

INTRODUCTION

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.

8.1 FOR LINE DETECTION AND FOLLOWING:

8.1.1 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.

8.1.2 1.2 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).

8.1.3 1.3 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

8.2 FOR BUILDING MAP

8.2.1 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.

8.2.2 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.

Chapter 9

Realistic constraints

9.1 COMPUTATIONAL-POWER-AND-RESOURCES

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.2 COMPLEXITY-OF-THE-SIMULATED-ENVIRONMENT

9.2.1 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.

9.2.2 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.

9.3 SIMULATING-SENSOR-DATA

9.3.1 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.

9.3.2 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.

9.4 ALGORITHMIC-CONSTRAINTS

9.4.1 a.-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.

9.4.2 b.-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 environ-

ment, 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.

9.5 SIMULATION-SOFTWARE-LIMITATIONS

9.5.1 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.

9.5.2 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.

9.6 SIMULATING-REAL-TIME-CONSTRAINTS

9.6.1 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.

Chapter 10

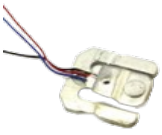
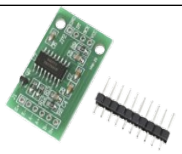
METHODS USED TO DESIGN THE PROJECT

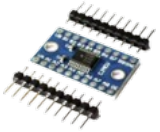
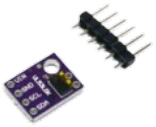
Chapter 11

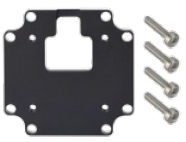
CONCLUSION AND FUTURE WORKS

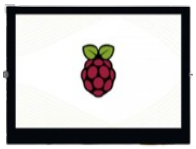
11.1 COST

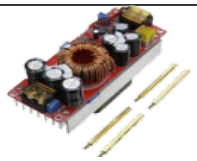
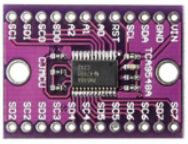
Table 11.1: here is the list of components

Components	Quantity	Price(per Unit)	ref
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 ₺	
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 ₺	
P Series Nema 23 Closed Loop Stepper Motor 2Nm(283.28oz.in) with Electromagnetic Brake	2	2971.78 ₺	
EG Series Planetary Gear-box 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 Vi-bration 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 ₺	
TP-Link TL-WR840N	1	569 ₺	
Raspberry Pi 4.3 Inch Capacitive Touch Screen DSI Interface 800”480	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 ₺	

WM-045 DC-DC 150W Voltage Booster	1	521.04 ₺	
Motorobit DC-DC 1500W 30A Voltage Boost Module	1	881.76 ₺	
TCA9548A I2C Çoklayıcı / Multiplexer Kartı	1	40.66 ₺	
3B Printer Limit Switch	2	37.07 ₺	
Drn956 16mm Acil - Stop Switch (Kafa 27mm)	1	145.08 ₺	

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

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] 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.

[7] R. L. Smith, "Open dynamics engine." `https://bitbucket.org/odedevs/ode/`, 2016. Online.

[8] R.-T. P. Simulation, "Bullet physics library." `http://bulletphysics.org/wordpress/`, 2016. Online.

[9] "Simbody: Multibody physics api." `https://simtk.org/home/simbody/`, 2016. Online.

[10] G. T. G. Lab and H. R. Lab, "Dart (dynamic animation and robotics toolkit)." http://dartsim.github.io/, 2016. Online.

[11] OGRE3D, "Object-oriented graphics rendering engine." http://www.ogre3d.org/, 2016. Online.

[12] 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.

[13] R. Wiki, "urdf/xml - ros wiki," 2023.