Jørgen Usterud Myrvold

Digital Twin of the KUKA KMR iiwa

Developing a Simulation Framework Using ROS 2 and NVIDIA Isaac Sim

Master's thesis in Engineering and ICT
Supervisor: Lars Tingelstad
January 2023

Jørgen Usterud Myrvold

Digital Twin of the KUKA KMR iiwa

Developing a Simulation Framework Using ROS 2 and NVIDIA Isaac Sim

Master’s thesis in Engineering and ICT
Supervisor: Lars Tingelstad
January 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Digital Twin of the KUKA KMR iiwa

Developing a Simulation Framework Using ROS 2 and NVIDIA Isaac Sim

Jørgen Usterud Myrvold

17-01-2023

Preface

This thesis concludes my master's degree in Engineering & ICT at the Norwegian University of Technology and Science in Trondheim. The project was conducted at the Department for Mechanical and Industrial Engineering in collaboration with the Robotics and Automation research group in the fall of 2022.

The project explores the possibilities a mobile manipulator's digital twin can provide and how it can be implemented. Creating a fully functioning digital twin is a significant endeavor, and this thesis aims to start this process and pave the way for further development.

I thank my supervisor Lars Tingelstad for his guidance and support throughout this project.

Jørgen Usterud Myrvold
17-01-2023

Abstract

Most robots today are used for repetitive and predictable tasks as autonomous operation in an ever-changing work environment is challenging to program. Moreover, developing such robotic applications is time-consuming and resource-demanding, but despite this, autonomous robots have become increasingly popular in recent years.

As computational power becomes readily available, robotic simulators can simulate the real world more accurately than ever. As a result, utilizing these powerful simulators can lower costs and reduce development time, resulting in faster robotic application development.

This thesis investigates the possibilities of a digital twin of the KUKA KMR iiwa developed in NVIDIA Isaac Sim using ROS 2. The KMR iiwa is a mobile manipulator consisting of the KMP 200 omniMove, a holonomic drive mobile platform, with the LBR iiwa 14 R820 manipulator mounted on top. The use cases for the robot span wide, and the mobile manipulator have unique capabilities that allow it to manipulate objects with high precision and to move over large surfaces. Programming such a robot is involved, and autonomous operation of the robot is desired.

This project aims to create a framework for efficiently developing, testing, optimizing, and deploying robotic applications. To achieve this, a physically accurate robot model must be developed. In addition, sensors and actuators will have to match the physical robot, and a standardized interface for controlling the physical and simulated robot is necessary.

Experiments focusing on the holonomic drive system, autonomous navigation, and mapping were conducted to evaluate the simulated robot's capabilities. In addition, the simulated robot has been tested in different environments to ensure the system performs as expected.

Two new ROS packages have been developed to control the robot, and two extensions for Isaac Sim were developed for importing and modeling the robot and creating an interface with the ROS system.

Sammendrag

De fleste roboter brukes i dag hovedsakelig til repetitive og forutsigbare oppgaver fordi det er utfordrende å programmere roboter som kan operere autonomt i uforutsigbare miljø. Det er tid- og ressurskrevende å utvikle, men til tross for det, har roboter i større grad blitt autonome de siste årene.

Når datamaskinene har blitt kraftigere har også simulatorer fått mulighet til å lage simuleringer likere virkeligheten. Ved å utnytte de nye, kraftige simulatorene kan man redusere kostnader og utviklingstid som igjen fører til raskere utvikling av robotapplikasjoner.

Denne oppgaven utforsker mulighetene en digital tvilling av KUKA KMR iiwa i NVIDIA Isaaac Sim har, ved bruk av ROS 2. KMR iiwa er en mobil manipulator bestående av en KMP 200 omniMove, som er en mobil plattform, og en LBR iiwa 14 R820, som er en manipulator montert på plattformen. Bruksområdene for en slik robot er vide og en mobil manipulator har unike egenskaper som tillater den å manipulere objekter med høy presisjon, i tillegg til å kunne bevege seg over store avstander. Det er komplisert å programmere en slik robot i detalj, så autonom operasjon er ønsket.

Dette prosjektet tar sikte på å utvikle et rammeverk for å effektivt utvikle, teste, optimere og *deploye* applikasjoner på roboten. For å gjøre dette må en detaljert modell av roboten utvikles. I tillegg må sensorer og aktuatorer nøyaktig simulere de faktiske instrumentene. Til slutt må et standardisert brukergrensesnitt utvikles for å kontrollere roboten.

Eksperiment fokusert på kjøre-systemet, autonom navigering og kartlegging av områder ble gjennomført for å evaluere egenskapene til den simulerte roboten. I tillegg ble den simulerte roboten testet i ulike miljøer for å forsikre oss om at den oppførte seg som forventet.

To nye ROS pakker har blitt utviklet for å styre roboten, og to utvidelser for Isaac Sim ble utviklet for å importere og modellere roboten, i tillegg til å sette opp et brukergrensesnitt mot ROS.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

1.1. Background and Motivation

Recent advances in computer science have enabled the fourth industrial revolution to focus on automation, machine learning, and real-time monitoring of production locally as well as across multiple production plants. With industry 4.0, every part of the supply chain is connected, providing opportunities for mass-produced, customized products and more efficient production.

One of the approaches during this transition is to further increase mechanization and automation [21]. There has already been a transition where robots have taken over tedious and labor-intensive tasks to free up human personnel to do other jobs. However, autonomous robots in Industry 4.0 have to be more versatile and able to adapt to unfamiliar tasks and situations. If this is accomplished, robots can do jobs only humans previously could do, increasing efficiency significantly.

One promising category of autonomous robots suitable for Industry 4.0 is mobile manipulators, such as KMR iiwa from KUKA. Such robots are recognized by their ability to move around on the factory floor while at the same time being able to move objects using their manipulator. Today robots are limited by not being able to take on more general tasks, but if this challenge can be overcome, these kinds of versatile robots will likely prove crucial during Industry 4.0

One of the main challenges of developing robotic applications for mobile manipulators is the cost of equipment and the time it takes. One proposed solution to decrease costs is to use experimentable digital twins for development, validation, and optimization [51]. However, this approach depends on the fidelity of the simulation and that it accurately simulates the robot and its environment. Recently there have been promising advances within computer science that enables simulators to have more realistic physics and photo-realistic visuals.

These advances in simulation technology show promise, and this thesis will explore how a digital twin can be used to develop robotic applications using the KUKA KMR iiwa. Furthermore, such a digital twin might also provide the possibility of developing machine learning- and computer vision tasks.

1.2. Problem Description

The main goal of this thesis is to create a digital twin of the KMR iiwa using NVIDIA's robotic simulator, Isaac Sim. The digital twin should enable developers to develop, test, optimize and deploy robotic applications more efficiently. To achieve this, a modular simulation framework has to be developed, and an accurate model of the robot has to be created with an accompanying ROS 2 interface. Furthermore, the ROS interface should provide relevant sensory information and the possibility to control the joints of the robot, both concerning driving the holonomic base and controlling the manipulator with its gripper.

1.3. Previous Work

This master's thesis concludes the work with a digital twin for the KMR started in the specialization project [31] written in May of 2022. This project explored the possibility of a digital twin and the prerequisites to make it a viable option for developing and optimizing robotics applications for the KMR. The project concluded that a digital twin could be realized and that Isaac Sim would be a promising simulator to achieve this. This is the basis for this thesis.

In 2020 Heggem & Wahl presented a ROS 2 stack for autonomous fetch and carry tasks for the KMR at the MANULAB at NTNU [10], and this was further improved in [5]. This work was successful, particularly in creating the ROS interface. However, the work did not focus much on the simulation, and most of the testing was conducted on the physical robot.

More work has also been done focusing more on simulation and the creation and application of a digital twin. For example, Fraunhofer IML created a robot for transporting pallets with holonomic drive and a digital twin for more efficient development and optimization of robotic algorithms [58, 14]. Another example is Anymal from ETH Zürich, which used the digital twin both to train its AI locomotion policy and to test the policies and algorithms in a safe, simulated environment [11]. Both of the robots were developed using NVIDIA Isaac Sim and are thus relevant for the development of the digital twin of the KMR.

1.4. Outline

This thesis is divided into five chapters. Chapter 2 presents the fundamental theory and necessary preliminaries, and Chapter 3 describes the implementation of the system and how different components tie together. Achievements and evaluation of the experiments conducted are discussed in Chapter 4 before further work, and a conclusion is presented in Chapter 5

Chapter 2.

Preliminaries

This chapter presents the fundamental concepts and topics required to understand the system presented later in the thesis. It also discusses decisions regarding the choice of technologies and frameworks and presents topics related to these frameworks.

Section 2.1 lays the foundation with a brief introduction of kinematics for wheeled robots, followed by Section 2.2, which describes the physical robot and its properties. Section 2.3 presents the theory behind mapping and navigation, and Section 2.4 describes the ROS system and relevant packages used in this thesis. Lastly, Section 2.5 discusses digital twins and robotic simulators, and Section 2.6 presents the NVIDIA Isaac Sim robotic simulator.

2.1. Robot Kinematics

The following sections 2.1.1 and 2.1.2 are primarily based on previous work in the specialization project [31], but is presented here as it forms the foundation of the kinematics of the mobile base. Subsection 2.1.1 presents basic terminology for describing robots in the plane, and Subsection 2.1.2 presents a kinematic model of a holonomic robot similar to the KMR.

2.1.1. Poses and Frames

A pose describes a position and orientation for an object in 2D or 3D space. Relative to a reference frame S, the pose of an object in 2D can be described as

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

In the case of a 2D plane, an object has 3 degrees of freedom (dof). It can move in the x and y direction, or it can rotate θ degrees around the z-axis. Hence the vector $\mathbf{X}$ is sufficient to describe any pose in 2D space. Even though $\mathbf{X}$ is sufficient to describe the pose of an object in 2D, it can be somewhat cumbersome to work with, and a more common description of poses is using homogeneous transformation matrices. A homogeneous transformation matrix, T, consists of a rotation matrix, R, and a column-vector, p, describing translation, as shown in Equation 2.2. The transformation matrix describes rotation and translation relative to a reference frame S.

$$T = \begin{bmatrix} R & p \\ \mathbf{0} & 1 \end{bmatrix} \tag{2.2}$$

In the case of 2D $p = [x, y]^T$ and R as described in Equation 2.3 [25, p. 64]. All rotation matrices $R \in SO(2)$ in 2D or $R \in SO(3)$ in 3D. This means that for all R it holds that $R^T R = I$ and $\det(R) = 1$ [25]. These properties imply that no scaling is involved in the rotation and that the rotation is uniquely defined.

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{2.3}$$

There are two main benefits of using this notation. The first is that it makes transformations between different frames into a matter of simple matrix multiplication, and the second one is that the procedure for transformations is the same in 3D. The only difference being that $R \in SO(3)$ and not $SO(2)$, and $p \in \mathbb{R}^{3\times1}$.

There are multiple ways of describing poses and frames, such as exponential coordinates and quaternions, which each have their use cases. Still, for a basic understanding of robotic control, this is sufficient.

2.1.2. Wheeled Robots

In many modern robotics applications, manipulators are placed atop mobile bases. This makes the robot more versatile but has some challenges when modeling its behavior. This section will discuss the kinematics of wheeled robots.

This section assumes that the robot moves on a plane surface with no skidding. This is to create a consistent model of how the wheels' rotation affects the robot's pose.

There are two main categories of wheeled robots; holonomic drive and non-holonomic drive. A formal definition is that a robot with holonomic drive has no constraints on its velocity $\dot{q} = (\dot{\theta}, \dot{x}, \dot{y})$, while a robot with non-holonomic drive has a single Pfaffian velocity constraint. An example of a non-holonomic robot can be a car where the constraint prevents the car from moving directly sideways, despite that it can reach any configuration (θ, x, y) in an obstacle-free environment [25, p. 514].

Even though a non-holonomic robot can reach any configuration in an obstacle-free environment, this is not always true in real life, where obstacles exist. The main advantage of holonomic robots is that they are more maneuverable. The ability to move directly sideways allows them to access spaces that non-holonomic robots can't access. In many cases maneuvering a holonomic robot is simpler than maneuvering a non-holonomic robot. There are, however, advantages to using non-holonomic robots as well. To achieve non-holonomic drive, the robot needs special wheels that can move in multiple directions, making them more fragile and unsuitable for higher speeds. Usually, holonomic robots use *omni wheels* or *mecanum wheels* shown in Figure 2.1. Both types have rollers with a small diameter which works best on flat, hard surfaces [25, p. 515].

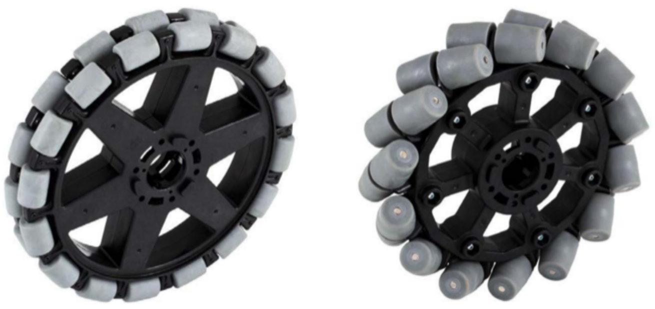


Figure 2.1.: Example of omni wheel (left) and mecanum wheel (right) [25, p. 514]

The rest of this section will focus on kinematics for holonomic robots, as the KMR has holonomic drive. Two main types of wheels are used in holonomic robots: omni wheels and mecanum wheels, shown in Figure 2.1. Both types allow the wheels to move sideways and in line with the wheel's direction. The omni wheels consist of multiple rollers around the wheel's circumference, which are oriented in line with the plane of the wheel. In contrast, the rollers on the mecanum wheels usually are mounted at about 45° to the plane of the wheel [25]. There are multiple ways of configuring a robot with omni wheels or mecanum wheels, but two common ones are shown in Figure 2.2

In Figure 2.2, the *driving* direction of the wheel is the direction in which the motor drives the wheel, and the *free sliding* direction is the direction in which the rollers

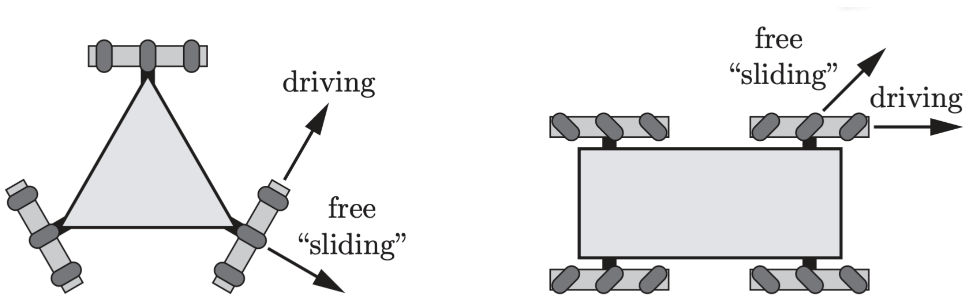


Figure 2.2.: Common configurations of omni wheels (left) and mecanum wheels (right) to achieve holonomic drive [25, p. 516]

can roll freely without any power from the motor. In this illustration, the rollers on the mecanum wheels are mounted at $\gamma = 45°$.

Kinematic Model for a Four Mecanum Wheeled Mobile Robot

Kinematics is used to describe the motion of a robot based on some input. The forward kinematics describes the robot's motion based on input from each joint's velocity, and the inverse kinematics describes the input to each joint or motor to achieve a particular movement. This chapter will discuss the kinematics for the mobile base in the 2D plane. Many modern frameworks provide integrated tools for controlling holonomic robots. However, a basic understanding of the parameters determining the kinematic model is necessary to tune the integrated tools properly. An example of such a tool is described in more detail in Subsection 3.2.3.

The KMR iiwa is a robot with holonomic drive using mecanum wheels, and Figure 2.3 illustrates the parameters to describe a mecanum wheel and the list below describes the parameters.

- X_R-Y_R, hereby references as X-Y, is the robot frame
- E-S is the wheel frame
- v_{ir} and ω_i is the angular velocity of the rollers and the wheel in the wheel frame (E-S).
- β is the angle between the S-axis and the X-axis.
- γ is the angle between the v_r and the E-axis.
- α is the angle between the X-axis and the line from O to the center of the wheel P.

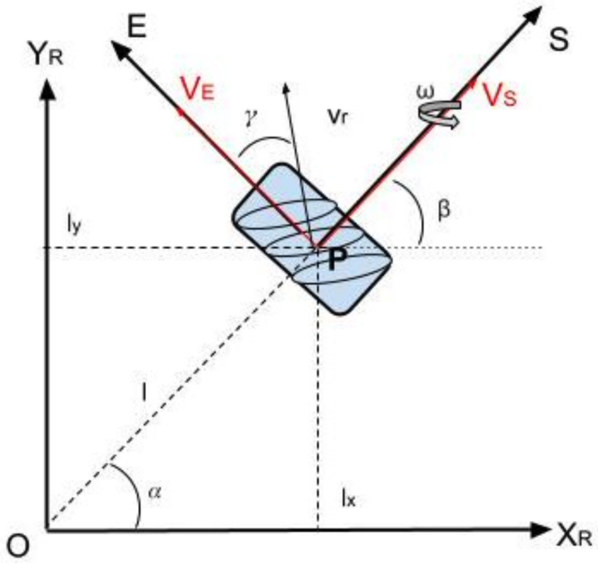


Figure 2.3.: Here, the variables used to derive the inverse kinematics for the robot are shown [9].

- l_y and l_x are the horizontal and vertical distances from the center of the robot.
- r is the radius of the wheel.

The details of the derivation of the kinematic model can be seen in [31] and will not be presented in detail here. The results are, however, of interest, and they are shown in Equation 2.4

$$\begin{aligned}
\omega_1 &= \frac{1}{r}(v_x - v_y - (l_x + l_y)\omega) \\
\omega_2 &= \frac{1}{r}(v_x + v_y + (l_x + l_y)\omega) \\
\omega_3 &= \frac{1}{r}(v_x + v_y - (l_x + l_y)\omega) \\
\omega_4 &= \frac{1}{r}(v_x + v_y + (l_x + l_y)\omega)
\end{aligned} \tag{2.4}$$

Equation 2.4 concludes the inverse kinematics problem for a robot with mecanum wheels mounted parallel to the direction of travel and with $\gamma = 45°$, which is the case with the KMR. From this, it is clear that the essential parameters are the longitudinal and lateral distances from the center of the robot to the wheels, l_x and l_y. The implementation of this in the simulated robot is further described in

Subsection 3.2.3.

2.2. KUKA KMR iiwa

The KUKA KMR iiwa is a mobile manipulator consisting of a mobile base, the KMP 200 omniMove, hereby referred to as the KMP, and a manipulator, the LBR iiwa 14 R820, hereby referred to as the LBR. Figure 2.4 shows the entire robot. This section will briefly present the main functionality of each of the parts and how they work together.

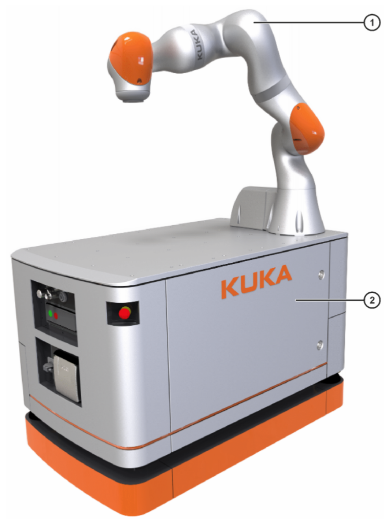


Figure 2.4.: The KMR iiwa. ① is the LBR iiwa 14 R820 and ② is the KMP 200 omniMove [17]

For this project, the robot is only used in the simulation. Thus implementation details of how to develop robotic applications for the physical KMR using Sunrise OS and the Sunrise Cabinet are not discussed here. A ROS 2 bridge to the Sunrise system was developed in [10, 5], and the interface developed here will, for the rest of this thesis, be considered the interface for controlling the robot.

2.2.1. KMP 200 omniMove

The KMP is the mobile base that transports the LBR and possibly other gods. The robot is equipped with four mecanum wheels, which allow the robot to move in any direction in the plane. The wheels and control unit enables the KMP to move with a precision of ±5 mm. Figure 2.5 illustrates the coordinate frame related to the KMP/KMR with the configuration of the wheels.

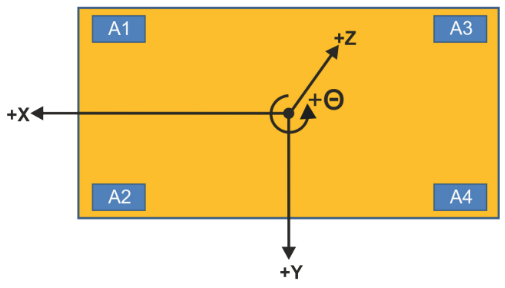


Figure 2.5.: Coordinate frame of the KMP. A1 through A4 denotes the mecanum wheels

The KMP has a maximum velocity of 1 m/s in the x direction, and a maximum velocity of 0.56 m/s in lateral and diagonal movements [17]. Moreover, the robot is surrounded by a protective field which, if violated, slows the robot down to lower speeds to avoid dangerous situations. The area of the protective field can be set using Sunrise OS.

To detect violations of the protective field, the KMP uses two SICK S300 safety laser scanners [17], located at B1 and B4, shown in Figure 2.6. Each scanner has a horizontal range of 270°, which surrounds the robot.

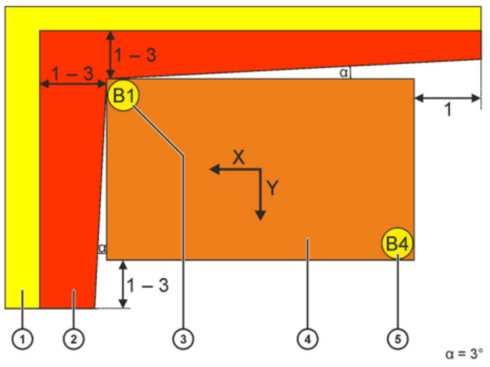


Figure 2.6.: Location of the laser scanners and protective field.

The laser scanners are mounted 150 mm above the ground and will only detect obstacles at this height. This is something to be aware of in a warehouse where the robot will not detect shelves, e.g., 0.5 m above the ground. If the robot drives under the shelf, it will collide without stopping, as the protective field was not technically violated.

2.2.2. LBR iiwa 14 R820

The LBR iiwa is a lightweight 7dof cobot designed to work alongside humans without needing a protective cage. The arm has torque and position sensors in each joint which communicate the robot's configuration. The joints and working envelope of the robot are shown in Figure 2.7

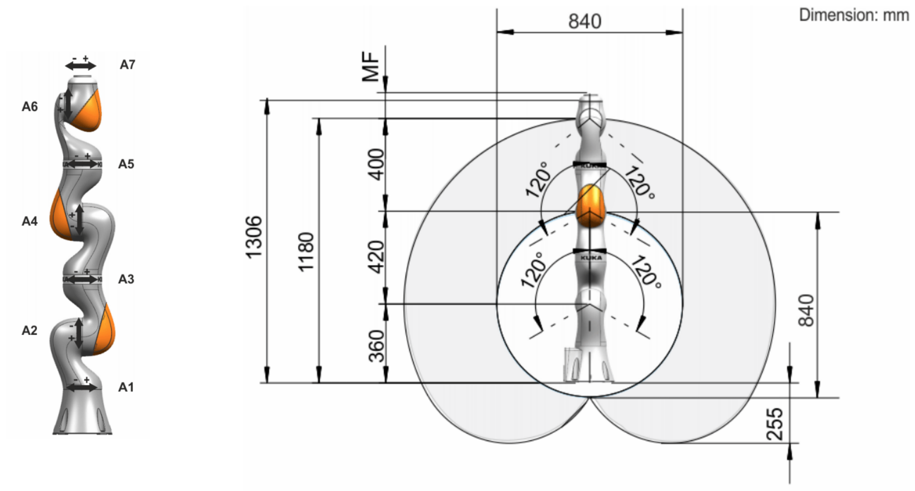


Figure 2.7.: Left figure shows the joints of the LBR, and the right figure shows the working envelope

In the context of the KMR, the LBR can be used to pick objects from a workspace, place them on the KMP, and deliver them to another workspace. With a precision of ±0.1 mm, the LBR is also suited for assembling tasks. The arm has a media flange, allowing different accessories to be mounted as end effectors. In addition, the media flange provides a power supply and data transmission to control various tools or grippers.

2.3. SLAM and Navigation

Autonomous navigation and mapping are two significant challenges using Autonomous Mobile Robots (AMRs). For a robot to move efficiently in an area, it needs a map. However, the creation of such a map can be challenging. Navigation of AMRs involves many problems, such as localization, path planning, and object avoidance. [47] mentions four core requirements to achieve such a task.

1. Map
2. Pose of the robot
3. Sensing

4. Path calculation and driving

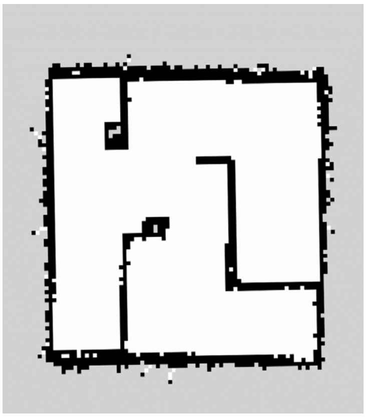
If the robot knows its pose and can sense its surroundings, it can create a map using SLAM. If the robot also has a map available, it can accomplish navigation tasks that take the robot from point A to point B. This chapter will consider the KMR as an AMR where navigation of the mobile platform is in focus.

This section is based mainly on previous work in the specialization project [31]; however, it forms a foundation for implementing SLAM Toolbox and Navigation 2 later. The section is condensed and includes only the most essential concepts. First, Subsection 2.3.1 describes maps and how to represent them, then Subsection 2.3.2 discusses how maps are created using SLAM, and lastly, Subsection 2.3.3 describes the process of navigation and path planning.

2.3.1. Maps

Traditionally a 2D occupancy grid map (OGM) has been used to represent maps, but in recent years 3D maps have been used and sometimes in combination with object segmentation [47, p. 324]. However, as the KMR is a mobile manipulator moving on a plane factory floor, a 2D map is sufficient for navigation purposes and will therefore be the focus of this section.

An OGM is represented as a greyscale image as shown in Figure 2.8a. Here the white area is where the robot is free to move, the black area is occupied, and the gray area is unknown.



(a) Example of an occupancy grid map

```
image: map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

(b) Example of `.yaml` file associated with the map

Figure 2.8.: Example of an OGM and associated `.yaml` file created by Turtlebot 3 [47, p. 325]

When computing the map, each pixel is assigned a value based on the probability that it is occupied, denoted as *occ*. This probability is calculated using the posterior probability of Bayes' theorem. If the value of *occ* is close to 1 indicates a higher probability that the area is occupied and if *occ* is closer to 0 indicates a free area. These values are published as ROS messages and converted to integers $[0, 100]$ including -1, which indicates that the area is unknown [47].

Accompanying the map is a configuration `.yaml` file, which specifies the properties of the map. The resolution specifies the area of a pixel in meters, and the origin is the 2D pose of the lower-left pixel on the form $[x, y, \text{yaw}]$ (yaw is often ignored) [7, 59].

An important addition to the OGM is the costmap used for navigation. This will be further discussed in subsubsection 2.3.3

2.3.2. SLAM – Simultaneous Localization and Mapping

As discussed in the previous section, maps are essential to AMRs, but they are not always available. Simultaneous localization and mapping, or SLAM, is the process of creating a map using a mobile robot with different sensors.

A formal definition of the SLAM problem is presented in [54].

> A mobile robot roams an unknown environment, starting at an initial location x_0. Its motion is uncertain, making it gradually more difficult to determine its current pose in global coordinates. As it roams, the robot can sense its environment with a noisy sensor. The SLAM problem is the problem of building a map of the environment while simultaneously determining the robot's position relative to this map given noisy data.

The challenge of SLAM is to do both localization and mapping simultaneously, whereas each task can easily be done separately.

A graphical representation of the SLAM problem is shown in Figure 2.9.

In Figure 2.9 $t \in [0, T]$ is the time interval, and T is the terminal time. x_t is the robot's pose at time t, and m is the static map and therefore time irrelevant. u_t is the relative motion of the robot, often derived from odometry, and z_t is the observations of the map gained from sensors. Together the SLAM problem can be defined probabilistically as

$$p(x_t, m | Z_t, U_t) \tag{2.5}$$

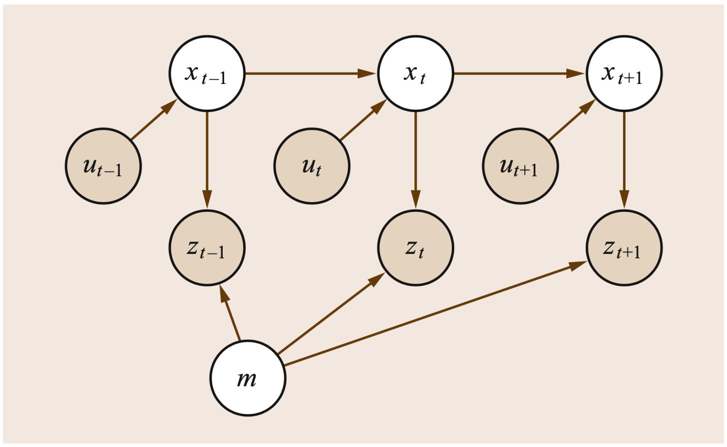


Figure 2.9.: Graphical representation of the SLAM problem [54, p. 1154]

Equation 2.5 describes the *online* SLAM problem, which is when the robot calculates its pose and the map in real time as it discovers the world. To calculate this probability, the robot needs a model that relates the odometry, u_t, to the location, x_t, and the observations, z_t, to the map, m. The two main models used are shown in Equation 2.6.

$$\begin{aligned} \text{Motion model:} \quad & p(x_t|x_{t-1}, u_t) \\ \text{Measurement model:} \quad & p(m|x_t, z_t) \end{aligned} \tag{2.6}$$

The motion model describes the probability that the robot is located at x_t given the last position, x_{t-1}, and the odometry in the previous interval u_t. In contrast, the measurement model describes the probability of doing the observation z_t given its pose x_t and the current map [54].

There are three main paradigms in SLAM used to calculate these probabilities and, finally, to create a map. The first SLAM applications used extended Kalman filters (EKF SLAM), but in most cases, EKF has been replaced by particle filters for online SLAM. These two approaches can be classified as *filtering approaches*. They rely on new measurements to refine the estimate of the state on the go and thus create a map. The last paradigm is graph-optimized approaches which were originally used for full SLAM but have recently seen adoption in online SLAM applications as well [54, 19]. Today it is the most common approach, and it will be explained in more detail in the following section.

Graph-based Optimization

Graph-based optimization techniques for SLAM were mainly used for full SLAM problems, but in recent years the technique has been adopted in online SLAM applications. The idea is that the robot's pose and landmarks' location are represented as nodes in a graph. Each pair of points, x_{t-1}, x_t, along the robot's path is connected with an edge containing information from the odometry u_t. The edges between the robot's location x_t and a landmark m_i describe the soft constraint between them [54]. Finally, this results in a sparse graph used to generate the map.

When the graph is created, the optimization problem is to minimize the error caused by the soft constraints. Different approaches exist for this, such as [18, 3].

One significant advantage of this approach is that the sparse graph allows much higher dimensional maps compared to EKF SLAM. Due to the limited space used to store and update the graph, graphical SLAM can create significantly larger maps than other methods.

2.3.3. Navigation

Navigation is the task of moving the robot from an initial pose to a goal pose using a map. The robot should be able to plan a path and avoid obstacles autonomously. The task consists of the following four subtasks [47]

1. **Sensing:** The robot gathers information from the wheel encodes and other sensors that describe the pose of obstacles, such as walls, furniture, and other objects relative to the robot.

2. **Localization / Pose estimation:** Based on the sensory input, the pose of the robot is estimated on the map. Multiple such methods exist, but one commonly used is AMCL which will be discussed in subsubsection 2.3.3.

3. **Motion planning / Path planning:** Creates a path from the current pose to the goal pose on the map. Usually, this task is separated into a global path which creates a path throughout the whole map, and a local path which handles path planning for a small area close to the robot.

4. **Move / Obstacle avoidance:** When the robot follows the planned path, obstacles and moving objects can occur. Obstacle avoidance is the task of avoiding collision with these objects and planning an alternative path. DWA is one algorithm used for this task, and it will be explained in subsubsection 2.3.3.

The following subsections will present three main components of navigation; costmaps, Adaptive Monte Carlo Localization (AMCL), and Dynamic Window Approach (DWA)

Costmap

As a result of SLAM, a static map for navigation is generated. It defines occupied areas, free areas, and unknown areas. A costmaps is an extension of this map describing the cost for the robot to move in different parts of the map. A higher cost indicates a higher probability of colliding. An example of a costmap is shown in Figure 2.10

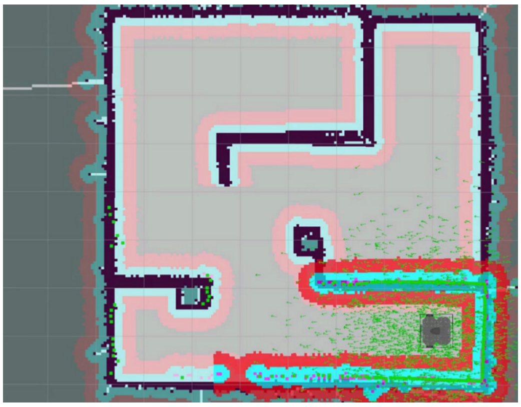


Figure 2.10.: Example of a costmap corresponding to the map in Figure 2.8a [47, p. 338]

The costmap is generated based on the OGM in Figure 2.8a, and the brighter colors are the parts of the map recognized by the robot from its current pose.

Costmaps can be separated into two types: a global costmap for navigation over the entire fixed map and a local costmap for obstacle avoidance and path planning in the area close to the robots. Both maps are represented the same way with a grid containing values between 0 and 255 which indicates the cost. Figure 2.11 shows the relationship between the distance to the object and the costmap value.

Using this map, different pathfinding algorithms can be used to generate a path. There exist multiple ROS packages and other implementations which automate the process of path planning. One example is the ROS package Navigation 2 [27].

Adaptive Monte Carlo Localization

Adaptive Monte Carlo localization (AMCL) is based on the Monte Carlo localization (MCL) pose estimation algorithm using particle filters. The main benefit

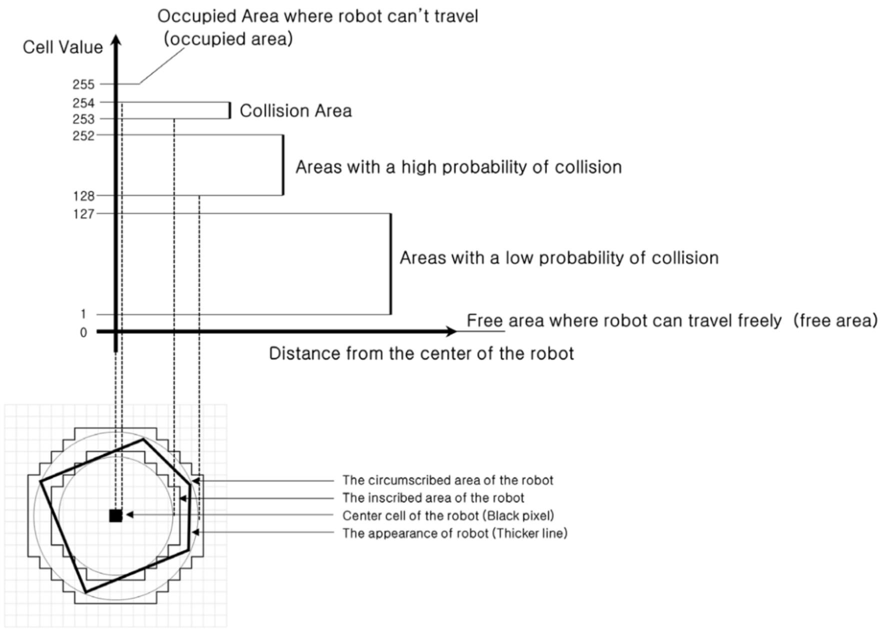

Cell Value
Occupied Area where robot can't travel
(occupied area)
255
254
253
252
128
127
1
0
Collision Area
Areas with a high probability of collision
Areas with a low probability of collision
Free area where robot can travel freely (free area)
Distance from the center of the robot
The circumscribed area of the robot
The inscribed area of the robot
Center cell of the robot (Black pixel)
The appearance of robot (Thicker line)

Figure 2.11.: Relationship between the distance to the object and the costmap value [47, p. 356]

of AMCL is that it achieves better real-time performance by using fewer samples (equivalent to particles) than MCL.

The localization procedure is similar to the general algorithm for particle filters which consists of predicting possible samples of the pose, updating the samples with sensor information, estimating the pose, and resampling, which creates new likely samples.

This algorithm is the default localization algorithm used by the Navigation 2 system [27].

Dynamic Window Approach

The Dynamic Window Approach (DWA) is a commonly used collision avoidance strategy in ROS. The robot initially follows a global path, but when obstacles occur that are not present on the map, the robot needs to plan a local path, and DWA is one strategy for planning this path.

The robot has a limited number of available movements based on the dynamics of the robot. A holonomic robot can move in x and y direction in addition to rotation θ around the z-axis. The task of DWA is to continuously choose the optimal collision-free velocity. DWA does not primarily consider the robot in the 2D xy-plane, but rather in the velocity space consisting of (v_x, v_y, ω), where ω is the rotational velocity around the z-axis. The algorithm searches the velocity space for possible velocities and chooses the best one.

To choose the best path, an objective function $G(v_x, v_y, \omega)$ considers different metrics and decides what combination of (v_x, v_y, ω) yields the best outcome. To choose the best path, the objective function may consider the cost on the costmap, distance to the goal, orientation to the goal, and orientation to the path [22]. When the best velocities are determined to maximize the objective function, the robot executes this.

2.4. ROS 2 – Robot Operating System

The Robot Operating System, or ROS [28], is a set of open-source software tools and libraries used for developing robotic applications. It contains robot-specific drivers, algorithms for robotic applications, and other tools to aid development [49].

This section will discuss different ROS-related topics, present relevant concepts and briefly explain some of the ROS stacks used in this thesis. Subsection 2.4.1 presents basic concepts of ROS necessary for understanding the system, while Subsection 2.4.2 briefly discusses the underlying communication protocol ROS uses. Subsection 2.4.3 presents URDF, the default method for describing robots using ROS, and Subsection 2.4.4 and Subsection 2.4.5 presents two ROS stacks used for mapping and navigation.

2.4.1. Introduction to ROS

ROS is not an operating system, as the name suggests, but rather a middleware aimed at tackling previously complex and inefficient development of robotic applications. Instead of writing software specific to one manufacturer, ROS provides a common interface for different robots [49]. The following paragraphs explain some basic concepts and terms used later to describe the system.

All computation in ROS is executed in *nodes*. A node handles computation related to a small part of the system, such as estimating the robot's pose, combining laser scans, publishing images from cameras, or other tasks. Instead of having large nodes that do many operations, it is considered best practice to have specific nodes that do particular tasks and communicates with each other using *topics*, *services*, or *actions*.

Communication over *topics* uses the publish-subscribe pattern where a laser scanner, for example, publishes a new message on a topic (e.g. /scan) each time a new scan is read. The standard ROS distribution provides multiple predefined message types containing specific data, but custom messages can also be created. All custom message types are defined using the interface definition language (IDL) and have file endings `.msg`. Topics are primarily used to distribute and consume continuous data streams such as sensor data or robot state.

Services are used to trigger function calls in other nodes. This can be used to execute quick calculations such as inverse kinematics or query a node's state. It is considered bad practice to use services for longer running tasks to avoid unwanted side effects for other nodes.

For longer running tasks, *actions* are the preferred mode of communication. This is because actions can trigger tasks and provide feedback during execution. Examples of where actions are appropriate include robotic movement or slow perception routines. The common property of such tasks is that they are all time-consuming and can provide feedback during execution.

2.4.2. RTPS and DDS

ROS uses DDS/RTPS as its middleware for communication. The middleware provides serialization and transportation of data and decentralized discovery of nodes. DDS, or Data Distribution Service, can be viewed as a layer on top of RTPS (Real Time Publish Subscribe). DDS leverages the RTPS protocol for network communication; however, DDS also provides advanced data-management features. Such features include type systems, different communication patterns, and types of quality of service to simplify real-time communication in distributed systems [49].

ROS supports different DDS implementations, with eProsima DDS being the default for ROS 2 Foxy. All supported implementation of DDS implements the abstract ROS middleware interface (rmw) [49]. Issues may occur if different installations and builds of ROS are used in different parts of the system, as environment variables may be configured differently. DDS profiles can be manually set, so that environment variables match when various installations of ROS are used. One such configuration file is presented in Appendix A. Because the simulator ships its installation of ROS and other packages are developed using a local installation of ROS, the DDS profile has to be set explicitly for the different ROS installations to work together.

2.4.3. Unified Robot Description Format – URDF and Xacro

URDF files are the standard format to describe robots in ROS. A URDF file starts with the opening *robot* tag, which defines the robot, which contains links and joints specified with *link* and *joint* tags. Each link has an *inertial*, *visual*, and *collision* tag, which describes the properties of the link. The *inertial* tag specifies the link's mass, the center of mass, and its central inertia properties, and is used to model the robot in simulators. Both the *visual* and *collision* tags require a geometry that defines the robot's volumetric properties. The same geometry can be used for both, however, a simpler geometry is often utilized in the *collision* tag to reduce computation time. A geometry can consist of basic shapes such as boxes, spheres, or cylinders, but a mesh is usually used to describe the robot more accurately. A mesh in URDF is a trimesh that consists of multiple two-dimensional triangles connected together to form a 3D geometry describing the robot [60].

The `joint` tag describes a joint's kinematic and dynamic properties, and it connects different links to form a complete robot. The joints can be of various types, but the relevant ones for the KMR are *revolute* joints and *continuous* joints. A revolute joint rotates around an axis and has a limited range of motion specified by the limit property. This type of joint is commonly used to model manipulators. A continuous joint behaves the same as a revolute joint but without the limits, which means it is suitable to model wheels on a mobile robot. All joints also have physical properties such as friction and dampening, and a safety controller which specifies parameters to operate joints safely. Lastly, joints also have a *mimic* tag, enabling joints to mimic another joint's motion. This can be useful for modeling grippers with multiple joints but only one degree of freedom. However, it has to be noted that not all robotic simulators support the mimic property of joints.

Xacro (XML Macros)

For complex robots such as the KMR, URDF files often become large and unclear as complexity grows. Xacro is an XML macro language and preprocessor used to simplify the generation of such files. The file format shares the specification described for URDF, so a plain URDF file can also be a Xacro file, however contrary to URDF, xacro provides features such as variables and expressions, macros, the possibility of including files and control structures [8]. This results in code being reused and more straightforward configuration of parts of the robot.

In some cases, such as with Isaac Sim, the simulator requires a description of the robot using only URDF. All xacro files can be converted to plain URDF files using the built-in ROS command line tool, `xacro`. In most ROS workflows, regular xacro files can be directly parsed without going around the URDF conversion.

2.4.4. SLAM Toolbox

SLAM Toolbox [26] is a ROS package for generating maps of unknown environments. It is the default SLAM vendor in ROS 2 and was also integrated into the Navigation 2 project. This section will not discuss the inner working of the SLAM Toolbox as this can be found in [26], but rather focus on its integration with ROS.

SLAM Toolbox is a feature-rich package, however, it only supports laser scan input and generates 2D OGMs. Macenski et al. [26] argue that laser scanners provide the most robust measurements and are, therefore, the preferred sensor for SLAM. Other libraries, such as [53], provide options for visual SLAM; however, this comes at the cost of the ability to only map smaller areas.

The package works straight out of the box with minimal configuration. However, it requires frames to conform to the REP105 [29] to function correctly.

2.4.5. Navigation 2

Navigation 2 (hereby referred to as Nav2) is a project containing several ROS packages to provide autonomous navigation for different types of mobile robots. This section will not go into implementation details of Nav2 as this can be found in [27], but rather the integration with the ROS system.

Nav2 works out of the box with minimal configuration. The project includes launch files for different applications and clear instructions on implementing the project into different systems. The expected inputs for Nav2 are listed below [57].

1. TF transforms conforming to REP 105 [29]
2. A map
3. A behavior tree XML file
4. Relevant sensor data sources

As output, Nav2 generates command velocities for both holonomic and non-holonomic robots.

The expected inputs listed above are specified in a parameter file provided to Nav2 in the launch file. For a robot with holonomic drive, other parameters must also be modified for the Nav2 to function optimally. The possible parameters are presented in [57].

2.5. Digital Twins

This section is primarily based on the previously written specialization project [31] and will briefly present some essential topics related to digital twins and robotic simulators. For a more in-depth analysis, readers are referred to [31].

One of the main challenges of developing robot applications for mobile manipulators, such as the KMR, is that testing is time-consuming and requires expensive resources. An essential criterion for the widespread adaptation of such robots is that development costs are lowered, and testing becomes less resource-demanding. Utilizing a *virtual testbed* is proposed in [51] to reduce the development cost and to streamline integration. A schematic representation of a virtual testbed is shown in Figure 2.12

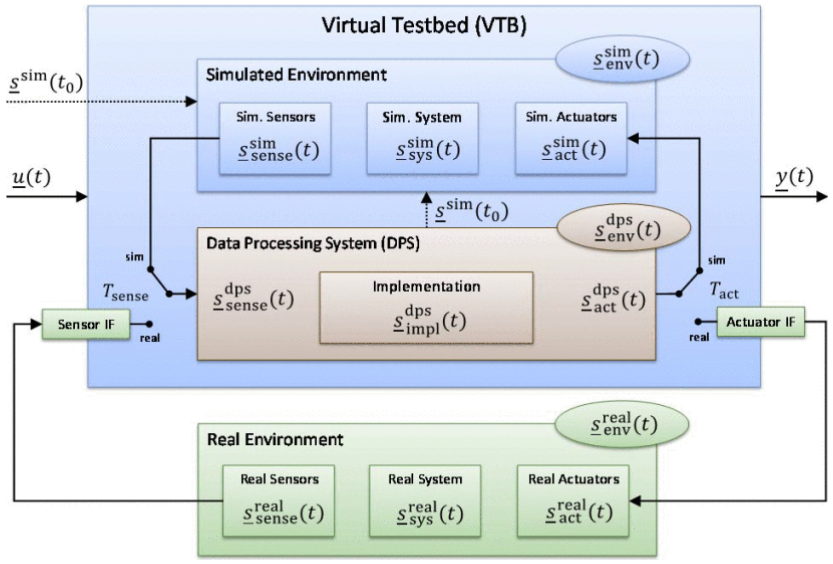

Virtual Testbed (VTB)
$\underline{s}^{\text{sim}}_{\text{env}}(t)$
Simulated Environment
$\underline{s}^{\text{sim}}(t_0)$
Sim. Sensors
$\underline{s}^{\text{sim}}_{\text{sense}}(t)$
Sim. System
$\underline{s}^{\text{sim}}_{\text{sys}}(t)$
Sim. Actuators
$\underline{s}^{\text{sim}}_{\text{act}}(t)$
$\underline{u}(t)$
$\underline{y}(t)$
$\underline{s}^{\text{sim}}(t_0)$
$\underline{s}^{\text{dps}}_{\text{env}}(t)$
Data Processing System (DPS)
sim
sim
T_{sense}
T_{act}
$\underline{s}^{\text{dps}}_{\text{sense}}(t)$
Implementation
$\underline{s}^{\text{dps}}_{\text{impl}}(t)$
$\underline{s}^{\text{dps}}_{\text{act}}(t)$
Sensor IF
real
real
Actuator IF
$\underline{s}^{\text{real}}_{\text{env}}(t)$
Real Environment
Real Sensors
$\underline{s}^{\text{real}}_{\text{sense}}(t)$
Real System
$\underline{s}^{\text{real}}_{\text{sys}}(t)$
Real Actuators
$\underline{s}^{\text{real}}_{\text{act}}(t)$

Figure 2.12.: Virtual testbed [51]

The virtual testbed focuses on developing the data processing system by enabling fast switching between the simulated environment and the real environment. The data processing system is usually the most complex part of the system. However, if a common interface for the sensors and actuators exists in the simulated and real environment, transitions between the two environments can quickly be done. In addition to streamlining development, a simulated world provides a safe environment where neither humans nor robots are exposed to hazards.

For such a system to work, a realistic robotic simulator must create a lifelike simulated environment. The following section discusses this issue.

2.5.1. Robotic Simulators

In the specialization project [31], a comprehensive comparison of different robotic simulators was conducted. This section will present a summary of this comparison.

In order to develop a DT and be able to test it, a simulator has to accurately simulate the real world. It needs a physics engine to replicate collisions between objects and to allow the robot to affect its environment.

[4] defines a robotics simulator as an end-user software application that includes at least the following functionality:

1. Physics engine for realistic modeling of physical phenomenon
2. Collision detection and friction models
3. Graphical User Interface (GUI)
4. Import capability for scenes and meshes
5. API especially for programming languages used by the robotics community (C++/Python)
6. Models for an array of joints, actuators, and sensors readily available

Table 2.1 shows an overview of common robotics simulators with some of their key features.

Table 2.1.: Comparison between popular robotics simulators [4]. (NVIDIA Isaac was not part of the original table from [4])

Simulator	RGBD + LiDAR	Force sensor	Linear + Cable actuator	Multi-Body Import	Soft-Body Contacts	DEM Simulation	Fluid Mechanics	Headless Mode	ROS Support	HITL	Teleoperation	Realistic Rendering	Inverse Kinematics
Airsim	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓, Unreal	✗
CARLA	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓, Unreal	✗
CoppeliaSim	✓	✓	Linear only	✓	✗	✓	✗	✓	✓	✓	✓	✗	✓
Gazebo	✓	✓	Linear only	✓	✗	Through Fluidix	Through Fluidix	✓	✓	✓	✓	✗	✓
MuJoCo	✓	✓	✓	✓	✓	✓	Limited	✓	✗	HAPTIX only	HAPTIX only	✗	✗
PyBullet	✓	✓	Linear only	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓
SOFA	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓, Unity	✗
UWSIM	RGBD only	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓, custom	✗
Chrono	✓	✓	✓	✗	✓	✓	✓	✓	✗	✗	✓	✓, offline	✓
Webots	✓	✓	Linear	✓	✗	✗	Limited	✓	✓	✗	✓	✗	✗
NVIDIA Isaac	✓[44]	✓[4]	✓[44]	✓[44]	✓[4]	Through Flex [45]	Through Flex [35]	✓[44]	✓[44]	✓[39]	✓[36]	✓, Omniverse	✓[4]

This comparison presents features generally desired in robotic development, however, for a mobile manipulator, the use case is narrower, and the desired features do not span as wide. A mobile manipulator is a combination of a mobile ground robot and a manipulator. Each group of robots has its own set of desired features, which is the basis for selecting an appropriate simulator.

The thorough comparison in [31] concluded that NVIDIA's Isaac Sim provided the necessary features for developing a DT of the KMR. Its photorealistic rendering is the main feature that sets it apart from other popular simulators like Gazebo. It is also a simulator in rapid development, and new releases are frequent. Its integration with ROS is sufficient, and its large sensor suite and built-in set of robots provide a solid foundation to build the DT.

2.5.2. Sim-to-Real

Simulators create a replication of the real world, however, they are never perfect. The gap between these two worlds is called the *sim-to-real gap*, or sometimes the *reality gap*. For the development of robotics applications, it is essential that this gap is small so that development and testing in the simulator accurately represent how the robot will behave in the real world. This is also important when developing robots using RL transferred onto physical robots.

This section presents a brief overview of the Sim-to-real problem based mainly on the previously written specialization project [31]. For a more thorough discussion of the problem, readers are referred to [31].

The sim-to-real problem is mainly related to RL applications; however, it represents a broader challenge related to simulating robots. Different simulators have different features, and the choice of simulators is vital to closing the sim-to-real-gap. In general, more realistic simulators will naturally yield better results. However, one might measure *more realistic* slightly differently based on the application. Suppose the application uses an autonomous vehicle mainly relying on cameras as sensory input. In that case, photorealism will be an essential feature, while a simulator used for training an AI for robotic grasping may place more importance on physically accurate simulations.

A mobile manipulator is a hybrid of different robots, and thus it requires a general-purpose simulation environment. Furthermore, it needs to be physically accurate to enable realistic testing of robotic grasping with the manipulator while at the same time providing the relevant sensors for AMRs. Cameras would be a natural sensor to utilize in a mobile manipulator application, and for this, a photo-realistic environment would significantly narrow the sim-to-real gap.

2.6. NVIDIA Isaac Sim

NVIDIA Isaac Sim [41] (hereby referred to as Isaac Sim) is a robotic simulator based on NVIDIA Omniverse [42] (hereby referred to as Omniverse). Omniverse is a comprehensive platform using Pixar's open-source Universal Scene Description, or USD for short, to build 3D worlds and simulate large-scale virtual environments.

Isaac Sim uses this extensible platform, extending it with robotic-specific tools and workflows, to create a high-fidelity robotic simulator [44].

This section presents some of the most important topics for understanding how Isaac Sim works and how to create robotic applications using this simulator. Subsection 2.6.1 presents fundamentals of USD, which is the backbone of Omniverse and Isaac sim, and Subsection 2.6.2 discuss some common workflows and features of Isaac Sim. Subsection 2.6.3 and Subsection 2.6.4 briefly present the ROS features included in Isaac sim focusing on OmniGraph and Isaac ROS.

2.6.1. USD – Universal Scene Description

Universal Scene Description, or USD for short, is a high-performance, extensible software platform for collaboratively constructing animated 3D scenes [56]. The software is created by Pixar Animation Studio, primarily for content creation and interchange between different creative tools. The software is open source, and because of this, it has seen wide adoption in fields such as manufacturing, robotics, architecture, and design.

A couple of terms are needed to understand how to work with USD. The starting point for every project is a *stage*. A stage is an abstraction of the scene graph described in a USD file. It presents a composed view of all elements in the root USD file and its *references*. The class `UsdStage` implements a stage, and it enables efficient data queries, which allows editing prims, properties, and metadata throughout the root file and its references [56].

A *prim* is the primary container object in USD and can contain other prims and, or properties. A prim defines a namespace on a scene, which means that prims can have the same name as long as they are under different prims. Prims may also contain schemas that describe what kind of data is present within the prim. All objects in a scene are considered prims and can be created and manipulated through the `UsdPrim` class [56].

After a scene is saved, it can be imported, or *referenced*, in another scene. This means that the default prim and all prims within its namespace are added to the new scene. This is useful for creating reusable prims such as mecanum wheels. However, it should be noted that when creating scenes that will be used as references, the default prim should not include physics scenes, environments, or lighting, as this also will be referenced in the new scene. This will result in multiple physics scenes and environments, which will cause issues [56].

API

Pixar has created an API for C++ and Python to script scenes and interact with them. Through the API, it is possible to create prims and cameras, add physics properties, and extract properties from objects during simulations. This can be used to log data from simulations such as a robot's pose or the wheel and joint velocities.

When using the Python environment included in the installation of Isaac sim (and implicitly Omniverse), the original USD API is available through the `pxr` python package. This can be used to interact with objects in the Omniverse scene. However, Isaac Sim applications are built with NVIDIA Omniverse Kit™[34], which also provides a simplified API specific for robotics. This API is meant as a supplement to the USD API, but it abstracts away the complexity of the USD API while also merging multiple steps into one for frequently used tasks [44]. Both APIs can be used interchangeably independent of each other.

2.6.2. Working with Isaac Sim

When developing with Isaac Sim, there are three main workflows: GUI, extensions, and standalone Python. The three workflows have pros and cons, but they are not exclusive and can be used in combination.

The graphical user interface, or GUI, is the most intuitive workflow and is suitable for assembling worlds and robots, and mounting sensors and cameras. The main drawback is that collaboration requires a more complex infrastructure than regular code files and version control tools such as git. The USD files generated are usually too large to be handled by git, particularly GitHub. Omniverse Nucleus [43] is a solution to the collaboration issue, but it requires a complex infrastructure.

All tools for Isaac Sim are built as extensions, and they provide the possibility of using both the GUI and Omniverse Kit to access all functionality of Omniverse. Extensions usually perform one specific task, and multiple extensions can be used simultaneously. They run asynchronously, enabling them to interact with the stage without blocking physics or rendering stepping. Code is usually executed by callbacks tied to physics steps, ticks in time, or stage events. Developing extensions is more involved, but Isaac Sim provides basic robotics applications which can be inherited to start an extension with basic functionality. Extensions are more complex but are recommended for real-time sensitive applications, which is the case with many robotic applications [44]. This is also the most common workflow when using ROS.

The last workflow is standalone Python. This is the recommended workflow for large-scale training in reinforcement learning and systematic world generation and modification [44]. In this workflow, Isaac Sim is launched via python scripts, which can manually control rendering and physics steps to give total control over the simulation. Creating custom standalone applications is more complex, but it yields greater control. It is possible to run in headless mode, which can be an advantage for training reinforcement learning systems.

2.6.3. OmniGraph

OmniGraph is a visual programming framework developed for Omniverse, and within Isaac Sim, it is the main engine behind the ROS and ROS 2 bridges. The purpose of OmniGraph is to easily connect functions from different systems within Omniverse [44]. It provides multiple pre-built nodes for different applications, such as robotics and ROS, in addition to the possibility of creating custom nodes. Developers can interface with OmniGraph both through the GUI and python scripts. As with the other workflows, the scripts provide more customizability but at the cost of being less intuitive.

Isaac sim provides a suite of nodes to OmniGraph developed for robotic applications. These nodes include articulation controllers, sensor reading, differential, and holonomic drive controllers, in addition to ROS-specific nodes. The ROS nodes are part of the ROS 2 bridge extension. They include nodes such as laser scanner publisher nodes, image and camera info publisher nodes, transform tree publisher nodes, joint state publisher and subscriber nodes, twist subscriber nodes, and others [38].

2.6.4. Isaac ROS

NVIDIA Isaac ROS is a collection of ROS 2 packages specifically developed for NVIDIA hardware that also integrates with Isaac Sim [40]. The packages include AprilTag detection and pose estimation, image pipelines, and segmentation, fleet management services from the mission client package, object detection, pose estimation, visual SLAM, and other functionality. If NVIDIA hardware is used as part of the robot configuration, this can enable tight integration between the hardware and software with great potential.

Isaac ROS is not yet a part of the system developed for this thesis due to version incompatibility not being resolved by NVIDIA before the release of Isaac Sim 2022.2.0 16. Dec 2022 [37]. All Isaac ROS packages are developed for ROS 2 Humble, and previous versions of Isaac Sim have only supported ROS 2 Foxy. Isaac Sim 2022.2.0 have beta support for ROS 2 Humble, which enables the packages to be integrated into the Isaac sim environment. Even though these packages are not a part of this thesis, they should be mentioned as they have the potential to improve the integration with the physical robot and the simulator. To some extent, this creates vendor lock-in, however, the Isaac ROS packages are open source and released under an Apache 2.0 license [40].

Chapter 3.

System Configuration

This chapter presents the solution developed and how the different parts are integrated. Section 3.1 presents the URDF model of the robot and all its parts, and Section 3.2 describes the simulation environment and the newly developed extensions in addition to how the sensors and actuators are implemented. Lastly, Section 3.3 presents the different ROS packages used to control the robot.

3.1. KUKA KMR iiwa Model

As discussed in Subsection 2.4.3, URDF is the standard way of describing a robot when working with ROS. However, URDF files often become large and difficult to work with as they grow in complexity. To overcome this problem, a modularized description of the robot was developed using xacro. The kmr_description ROS package [32] contains the robot's description and visualization scripts. All directories and subdirectories referenced in this chapter are relative to the kmr_description package.

The configuration of the robot used in this project is based on Heggem & Wahl's work [10] as this was a successful configuration. In addition to the KMR, which consists of the KMP base and the LBR arm, a Robotiq 2F-85 gripper [50] was mounted at the end of the arm. Four Intel RealSense Depth Camera D435 [12] were mounted on the front, on the left and right side of the KMP base, and on the end effector of the LBR, respectively. To mount the gripper and the cameras, separate adapters also had to be mounted to make them compatible.

The reason for using xacro to develop the URDF description of the robot is that each component can be described as individual macros in their separate files to modularize the code. In the kmr_description package, the `urdf/` directories contain sub-directories for each component. Here the `urdf/robot/` folder contains the main robot, which combines all the other components, or macros, into one main xacro file, which is later converted into a URDF description. The kmr_description package also contains a plain URDF file of the robot, but this is generated based on the xacro file. A plain URDF file is needed, as Isaac Sim only has tools for importing URDF models and not xacro.

The following paragraphs describe the different components, or xacro macros. Where relevant, they discuss the implementation in more detail and discuss some limitations of the macros. All the macros are placed in their corresponding folders and are named `<macro_name>.xacro`. For most of them, the folder also includes a xacro file initializing the macro for visualization named `<macro_name>_standalone.xacro`.

3.1.1. Intel RealSense d435 Depth Camera Description

The description of the camera is located in `urdf/d435/`, and it is based on Intel's URDF description [48], which provides a comprehensive model. The macro describing the camera is named `sensor_d435` and is found in `d435/_d435.xacro`. This macro has multiple parameters for customizing it to the robot. The most important one, apart from `parent` and `*origin`, is the `use_nominal_extrinsics` parameter. A physical RealSense camera can publish its extrinsics at the camera_info topic, however, this is not possible directly in Isaac Sim. Therefore, to create the most accurate model of the camera, this parameter can be enabled to provide the extrinsics in the simulation. Enabling the `use_nominal_extrinsics` parameter includes the frames for each of the cameras in the URDF description, which allows cameras to be mounted in the exact location in the simulator. The camera with its frames is shown in Figure 3.1.

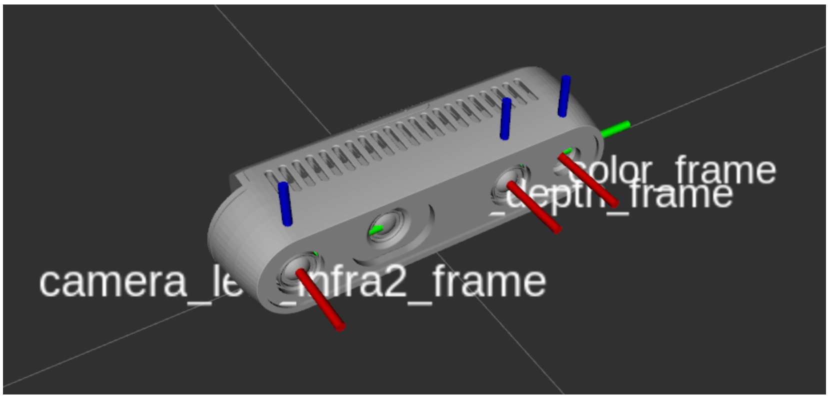


Figure 3.1.: URDF model of the Intel RealSense d435 visualized in rviz with a small selection of the camera frames. The frames from left to rigth are `camera_infra2_frame`, `camera_depth_frame` and `camera_color_frame`.

The model contains multiple other frames, but they are left out of the figure for clarity. The complete list of frames is shown in Table 3.1 and conforms to REP 103 [6].

Table 3.1.: List of the links/frames included in the camera description. All links are prefixed with the `name` attribute, which by default is "camera". Indented links are children.

Link	Description
bottom_screw_frame	Top level frame
link	Main link of the camera. Same as color_frame
color_frame	Frame for color camera with z up
color_optical_frame	Optical frame for color camera with z down
depth_frame	Frame for depth camera with z up
depth_optical_frame	Optical frame for depth camera with z down
infra1_frame	Frame for infra camera 1 with z up
infra1_optical_frame	Optical frame for infra camera 1 with z down
infra2_frame	Frame for infra camera 2 with z up
infra2_optical_frame	Optical frame for infra camera 2 with z down

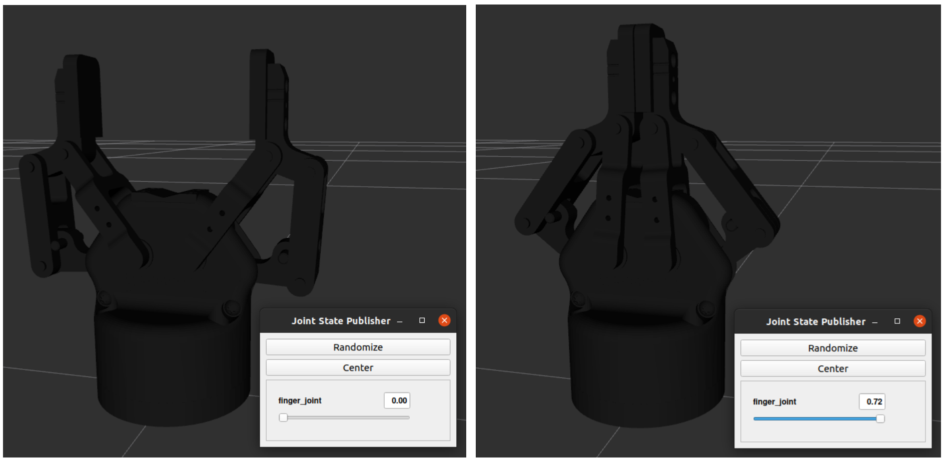
3.1.2. Robotiq 2F-85 Gripper Description

The description of the gripper is based on the work in [46]. One of the main challenges of describing the gripper is to correctly model all the joints. The gripper is a complex component with multiple joints, however, it only has one degree of freedom, which describes to which degree the gripper is open or closed. To make all the joints in the gripper move as intended with only one parameter controlling the openness of the gripper, the mimic attribute of joints is used. This forces the joint to mimic the main joint's articulation but with the possibility of setting a multiplier and offset. The value of the joint is then calculated as $\text{value} = \text{multiplier} \times \text{other\_joint\_value} + \text{offset}$.

By making all the joints mimic the `gripper_finger_joint`, the gripper correctly sets the position of all joints when setting the value of the `gripper_finger_joint`. An image of the gripper description is shown in Figure 3.2.

3.1.3. LBR Description

The description of the LBR arm was retrieved from [55] and is left unchanged. The macro is more complex than the others as it includes a separate transmission macro, `iiwa14/iiwa.transmission.xacro`, and a gazebo macro, `iiwa14/iiwa.gazebo.xacro`. The transmission macro is used to describe the relationship between actuators and joints. In this case, it is not strictly necessary,



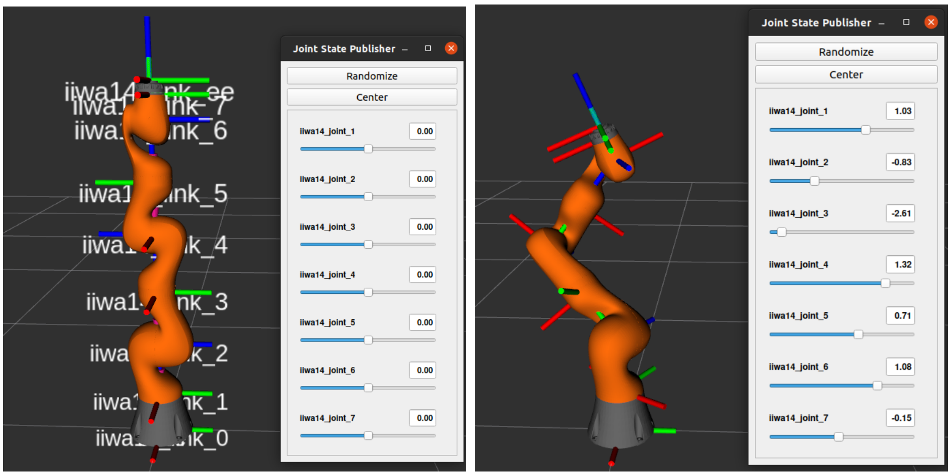
finger_joint 0.00

finger_joint 0.72

Figure 3.2.: Gripper in the open (left) and closed (right) position

however, it makes the model more flexible and provides the possibility of more customization. It is left in the repository for this reason.

The gazebo macro is specific for the Gazebo simulator and does not affect the model using Isaac Sim. This macro provides the possibility for more customization and is left in the repository even though it is not used in this case. Figure 3.3 shows the LBR in Rviz.



iiwa14_joint_1 0.00
iiwa14_joint_2 0.00
iiwa14_joint_3 0.00
iiwa14_joint_4 0.00
iiwa14_joint_5 0.00
iiwa14_joint_6 0.00
iiwa14_joint_7 0.00

iiwa14_joint_1 1.03
iiwa14_joint_2 -0.83
iiwa14_joint_3 -2.61
iiwa14_joint_4 1.32
iiwa14_joint_5 0.71
iiwa14_joint_6 1.08
iiwa14_joint_7 -0.15

Figure 3.3.: LBR manipulator in the starting position (left) and a random position on the right

3.1.4. KMP Description

The KMP macro is also based on [55] and is mostly left unchanged. In addition to the kmp200 macro, the folder contains a gazebo-specific macro. Again this is specific for the Gazebo simulator and allows the robot to move sideways to simulate holonomic drive with the mecanum wheels. The holonomic movement is achieved using the `force_based_move`-plugin from the hector_gazebo ROS package [15]. This workaround is used as the mecanum wheels in the URDF do not actually model the wheels with the rollers on the outside, but only an ordinary wheel with the visuals of a mecanum wheel.

The lack of properly modeled mecanum wheels is one of the major limitations of the simulation of the KMP and, therefore, also the movements of the entire robot. It is possible to drive the robot in a straight line back and forth as the collision mesh surrounding the mecanum wheels is more similar to a regular wheel. However, it is not possible to use the included wheels to model the holonomic movements, as the original model behaves similarly to a car with no steering input.

Previously there have been successful attempts at modeling mecanum wheels using URDF [1, 61], however, these methods were based on CAD models of the actual wheels. Unfortunately, KUKA provides no such CAD files so modeling the wheels exactly proves difficult.

For this thesis, a workaround utilizing both URDF and Isaac Sim's Universal Scene Description (USD) was used. The majority of the robot was modeled using URDF, except the mecanum wheels, which were modeled using USD. The wheels used on the KMP are the same as on O3dyn [58], an open-source holonomic robot with mecanum wheels. The wheels were scaled down to approximately fit the dimensions of the original wheels of the KMP. Figure 3.4 shows the original wheels from O3dyn and Figure 3.5 shows the original and scaled dimensions of the wheels

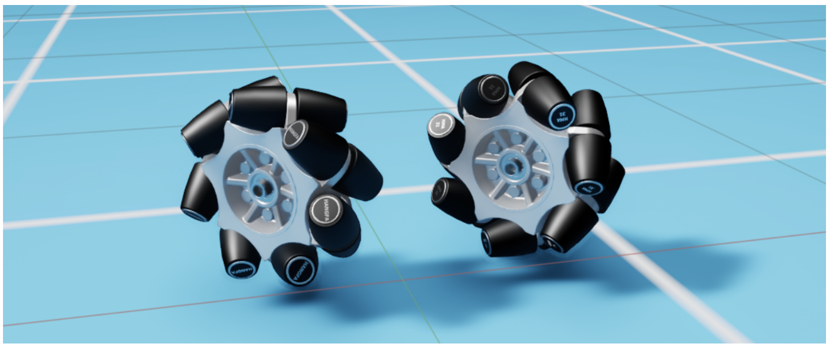


Figure 3.4.: Mecanum wheels from O3dyn [58]

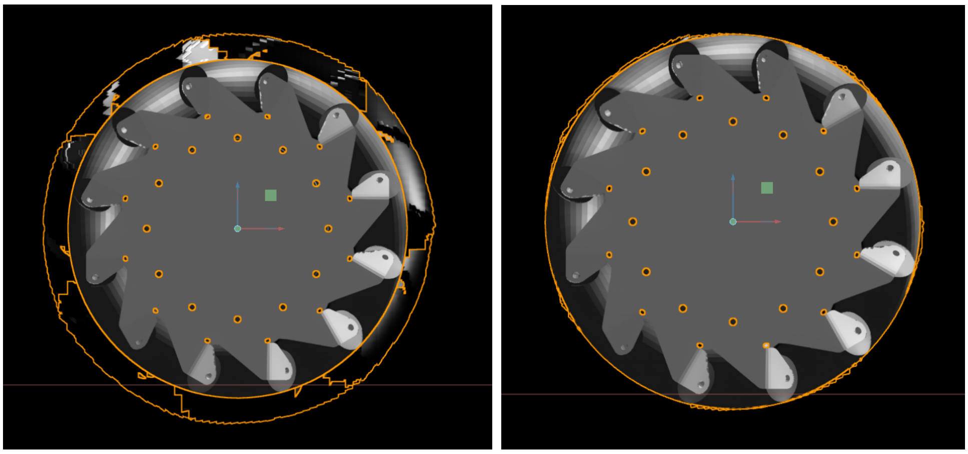


Figure 3.5.: The pictures are orthographic projections of the sides of the wheels. On the left, the outer circumference shows the original scale of the O3dyn wheels, and the inner circle shows the KMP wheel. The right figure shows the wheels scaled

To more easily add the USD mecanum wheels in Isaac Sim, an option for excluding the wheels was implemented for the KMP macro, which was used in the final model for the robot. The USD mecanum wheels were joined with the KMP using the KMR loader extension for Isaac Sim, which will be presented in Subsection 3.2.1. The configuration of the wheels overlaid with the original wheels is shown in Figure 3.6.

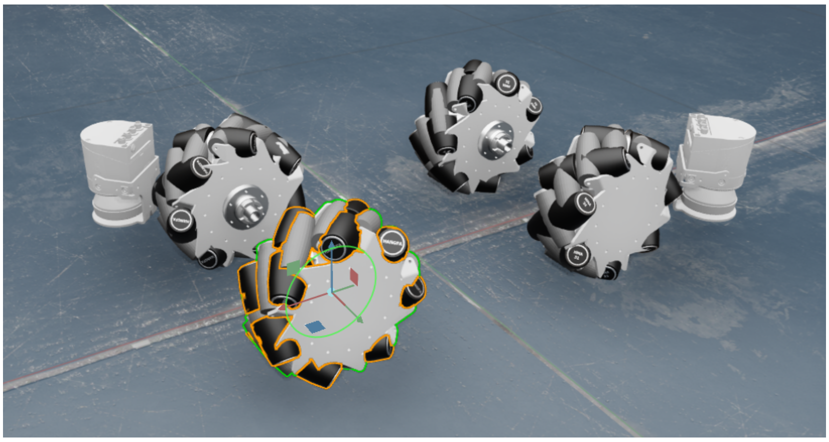


Figure 3.6.: Configuration of the mecanum wheels (black rollers) with the original wheels in grey. The diagonal boxes illustrate the laser scanners. The original wheels are removed for the final model.

3.1.5. Summary of the Robot Description

The kmr_description package, in combination with the KMR loader extension for Isaac Sim (Subsection 3.2.1), provides a good model of the robot using Isaac Sim. The model adheres to the REP 105 standard for ROS [29] and has most of the functionality the physical robot has. The main limitation of the model is the mecanum wheel and the holonomic drive. The dimensions here are likely not exactly correct, and the wheels lack suspension, which should be implemented to make the drive smoother. In addition, physics parameters, such as max joint torque, stiffness, and dampening, should also be tuned to match the robot's physical properties.

3.2. Simulation Environment

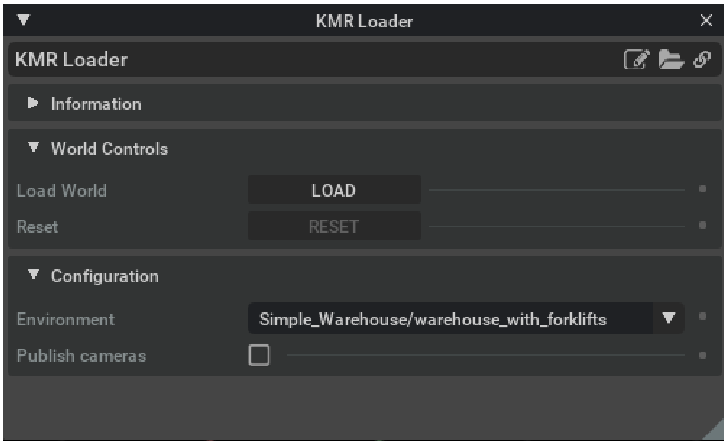
For this project, two new extensions [33] for NVIDIA Isaac Sim were developed. The *KMR Loader* extension loads the KMR into the simulator in the selected environment, and the *Pose logger* extension was developed to gather data from the simulator during experiments. This section will also discuss how sensors and actuators are implemented in Isaac Sim.

3.2.1. KMR Loader Extension

The *KMR loader extension* leverages the extension workflow to load the KMR into the selected environment. The extension loads the URDF generated from the ROS kmr_description package (Subsection 3.3.1) and rigs the robot with all sensors and actuators. In addition, the extension interface allows users to select which environment the robot should be loaded into and whether the simulation should publish images from the four cameras mounted on the robot. The interface is shown in Figure 3.7.

The extension also configures all ROS connections through the OmniGraph API. As of now, this includes

- Subscription to twist messages from the `/cmd_vel` topic to drive the robot
- Publish joint states and subscribe to joint commands for the manipulator
- Publish TF transforms and odometry
- Publish laser scans from two laser scanners
- Publish RGB and depth images from the four cameras
- Publish simulation time



Load World
Reset
Environment: Simple_Warehouse/warehouse_with_forklifts
Publish cameras: ☐

Figure 3.7.: KMR Loader extension interface

Each of the functionalities listed above is implemented in separate OmniGraphs, which can be inspected and edited in the simulator GUI or in the extension source code.

The option for not publishing camera data from the four cameras is included due to performance bottlenecks. When publishing data from one camera, the frequency of the images published is close to 60 fps which is the rate at which the simulator renders the scene. However, when using all four cameras, it only manages to publish images at a rate between 12-18 fps. All cameras publish both an RGB image and a greyscale depth image, resulting in a total of 8 image steams when all cameras are running simultaneously.

The extension is supposed to streamline the process of importing and rigging the robot in the simulator. It sets the KMR as the default prim in the stage, which allows the robot to be easily exported as a USD file that can be imported into any environment, also environments that are not included in the extension interface.

3.2.2. Pose Logger Extension

The *Pose Logger extension* is a tool for logging relevant data related to the KMR during simulation. The interface is simple and is shown in Figure 3.8. It initializes the pose logger, sets the KMR as the target prim, and exports saved data as a JSON file.

As of now, the extension logs the following data

- Timestamp
- Transformation matrix, $T \in SE(3)$, relative to the world frame

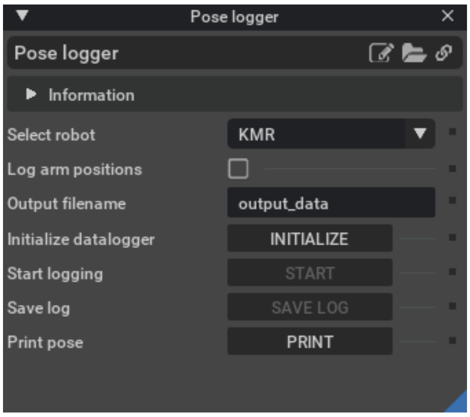


Figure 3.8.: Pose logger extension interface

- Wheel velocities
- Joint angles for the manipulator

The extension utilizes the Pixar USD API [56] to extract properties of the robot, and Isaac Sim's DataLogger object [44]. The extension is limited only to logging data from the KMR and the O3dyn robot, however, it is modular by design, and the source code can easily be modified to function with any robot.

The same extension repository [33] also includes a `data_processing` directory used to process and plot data. The python `Preprocessor` class extracts relevant data from the JSON data file and organizes the data as NumPy arrays for easier data manipulation and plotting.

3.2.3. Sensors and Actuators in Isaac Sim

The *KMR Loader* rigs the robot with different sensors and actuators to resemble the physical KMR. The following paragraphs will discuss the implementation of the cameras, lidars, and the holonomic drive, which are considered the major sensors and actuators in the KMR.

Cameras

For the KMR, four cameras were created to publish video streams to ROS. On the physical robot, Intel RealSense D435 has been used, which provides RGB-D images. In the simulator, a simplification has been made with the four cameras publishing two separate video streams, one RGB and one greyscale depth video.

The cameras are mounted at the *link* frame of the camera, which overlaps with the *color_frame*.

To publish the video streams, the `ROS2CameraHelper` node was used. It simplifies the procedure of publishing camera info and images as it takes an input viewport and a topic name for the video to be published on.

To better simulate the RealSense cameras, the cameras should be modeled as stereo pairs and utilize the comprehensive Intel RealSense ROS library [48]. This would allow more precise simulations providing point clouds and other functionality to closer mimic the physical cameras.

The framework provided in the KMR Loader makes modifications like this easy to perform. New camera parameters can be implemented as well as modifying existing configurations.

Lidars

Isaac Sim provides multiple range sensors for generating point clouds, regular 2D laser scans, and custom range sensors. As the KMR uses regular laser scans, two rotating laser scanners were mounted on the robot with the following parameters

Table 3.2.: A selection of important lidar parameters

Parameter	Value
Minimum range	0.4 m
Maximum range	30 m
Horizontal fov	246° (B1) and 249° (B4)
Horizontal resolution	0.4
Rotation rate	20 Hz

The horizontal fov is the range in degrees the laser scanner scans. In the specifications for the scanners [16], this range is 270°, however, in simulation, this caused the scanner to hit the robot and thus detect "objects" that were indeed the robot. Experiments showed that 246° and 249° for scanners B1 and B4, respectively, were the max ranges possible without the scans colliding with the robot. Other parameters can be set in the KMR Loader source code to accurately simulate the real laser scanners.

To publish the laser scans, the *IsaacReadLaserBeams* node and the *ROS2PublishLaserScan* node were used. The KMR Loader creates one OmniGraph for each of the laser scans. Both scans have been tested separately with no issues, and they both work with the kmr_concatenator presented in Subsection 3.3.2.

Honomic Drive

As discussed in Subsection 2.1.2, a holonomic drive robot needs a kinematic model to control its movement. For this thesis, Isaac Sim's *HolonomicController*, which is part of the `omni.isaac.wheeled_robots` [38] extension, is used. This controller was implemented as part of the OmniGraph interfacing with the ROS `/cmd_vel` topic. The node receives x, y, and θ inputs describing the direction of travel and outputs joint velocities to the *IsaacArticulationController* [38], which moves each individual wheel.

The kinematic model used in the *HolonomicController* depends on multiple parameters to calculate the correct joint velocities for each wheel, and they are listed in Table 3.3

Table 3.3.: Parameters for the holonomic controller

Parameter	**Value**
upAxis	`[0, 0, 1]`
wheelAxis	`[0, 1, 0]`
wheelRadius	`[0.1289999932 * OMNIWHEELS_SCALING_FACTOR] * 4`

wheelOrientations	`[[-0.5, 0.5, 0.5, -0.5],`
	`[ 0.5, 0.5, 0.5, 0.5],`
	`[-0.5, 0.5, 0.5, -0.5],`
	`[ 0.5, 0.5, 0.5, 0.5]]`

wheelPositions	`[[ 0.28, 0.1825, -0.2],`
	`[ 0.28, -0.1825, -0.2],`
	`[-0.28, 0.1825, -0.2],`
	`[-0.28, -0.1825, -0.2]]`

mecanumAngles	`[-135, -45, -45, -135]`
maxLinearSpeed	`1.0`
maxAngularSpeed	`0.51`
angularGain	`-1.0` (Rotation is opposite)

Here the wheel orientations are described as quaternions as that is the default in Isaac Sim [44]. For the wheel radius parameter, the original radius used for the O3dyn robot was used and multiplied with the `OMNIWHEELS_SCALING_FACTOR` to match the radius of the wheels on the KMR. This scaling factor is the same one used to scale the actual USD model of the wheels mounted on the KMR.

There have been made simplifications as the center of gravity of the KMR is not precisely centered, however, it works satisfactorily due to the significant weight differences between the KMP and the LBR. The parameters are specified in the KMR Loader extension and can only be edited in the source code and not in the simulator GUI.

3.3. ROS Setup

For this thesis, two new ROS packages were developed. Other previously developed packages for the KMR and standard ROS packages have been used. This section describes the implementation of ROS in the simulated environment.

The two newly developed ROS packages are the kmr_description package and the kmr_bringup package. In addition, the kmr_concatenator from [5] has been implemented into the system. These three packages will be presented in the following sub-sections.

3.3.1. kmr_description

The kmr_description package contains comprehensive descriptions of the KMR and its separate components. The package includes mainly xacro files used to visualize the robot using Rviz2 and to model the robot in simulation. These files have been described more in detail in Section 3.1. The package is based on [55], and modifications and additions have been made.

In addition to the URDF files describing the robot, the package contains a Rviz configuration file and a launch file for visualizing the robot and performing simple joint manipulation. The launch file `display.launch.py` includes launch descriptions for Rviz2, Joint State Publisher [23], Joint State Publisher GUI [24] and Robot State Publisher [20] and thus depend on the aforementioned packages. All the packages are well established within the ROS community and provide assistance for debugging problems with the URDF description.

The Robot State Publisher publishes a transformation tree describing all link's positions relative to one another. It is a known problem that this can cause issues when running simultaneously as Isaac Sim, which also publishes a transformation tree based on the position of the links within the simulator. However, as the kmr_description package is mainly used to describe the robot and other packages do not depend on it running simultaneously, this does not cause major issues. The workaround for visualizing the robot in Rviz while also simulating the robot is to use the transformation tree published by Isaac Sim and only launch the joint state publisher and not the robot state publisher.

3.3.2. kmr_concatenator

The kmr_concatenator package was originally developed in [5] and has been reused in this configuration. As SLAM Toolbox requires only one laser scan topic as input, and the KMR has two scanners on opposite corners of the base, these scans must be merged. Figure 3.9 shows a schematic description of how the kmr_concatenator package works.

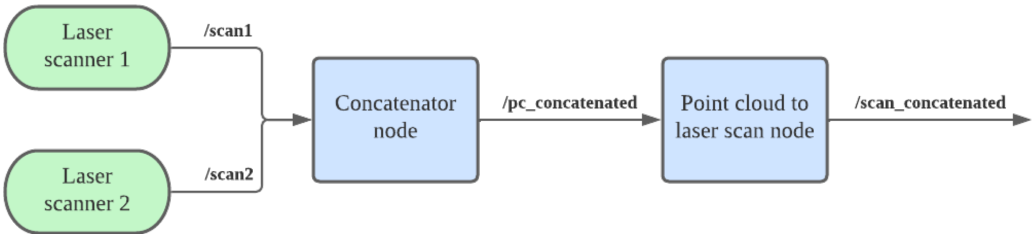


Figure 3.9.: Schematic description of the concatenation of two laser scanners. The green boxes represent the hardware publishing data, and the blue boxes are ROS nodes processing data. The arrows are labeled according to topic names

The package includes a launch file that launches the concatenator_node and a pointcloud_to_laserscan_node from the pointcloud_to_laserscan package [2]. In short, the concatenator_node reads laser scan messages from the two scanners at approximately the same time, converts them to point clouds, concatenates the point clouds, and publishes the concatenated point cloud. The pointclout_to_laser_scan_node converts the concatenated point cloud published on `/pc_concatenated` and publishes the converted point cloud as a laser scan to the topic `/scan_concatenated`. A more in-depth description of the package can be found in [5].

3.3.3. kmr_bringup

The kmr_bringup package is used to start the different nodes needed for different applications. The package includes three launch files for different applications and accompanying configuration files.

`bringup.launch.py` initializes the ROS interface for the robot, and it is the basic launch configuration. It bears similarities to the launch file in kmr_description (Subsection 3.3.1) but does not include the robot_state_publisher node because of the issues described in Subsection 3.3.1. `bringup.launch.py` launches the joint_state_publisher_gui to control the joints manually and Rviz to visualize the robot's state. Using this launch file, the robot can be driven, and its state with transformation tree, odometry, and laser scanners can be inspected in Rviz.

SLAM Toolbox Implementation

`slam_toolbox.launch.py` is the launch file used for mapping tasks. The launch file requires that the simulation and the kmr_concatenator are running. The kmr_concatenator has to initialize for about 5 seconds listening to the TF messages before the SLAM system can be launched. To address the timing problem, a lifecycle node could be used, however, due to time constraints, this has not been implemented.

For the experiments conducted, the `online_async_launch.py` launch file included in the SLAM Toolbox package was used in combination with the parameter file `config/slam_params_online_async.yaml`. The parameter file mostly used the default settings apart from specifying the scan topic. It is, however, possible to tune different parameters for improved performance.

Navigation 2 Implementation

The Navigation 2 project works rather well out of the box. One launch file was produced for this project with two different parameter files specifying the Nav2 parameters for a differential drive robot and a holonomic drive robot, respectively. Before launching `navigation.launch.py`, the simulation and the kmr_concatenator have to run. The launch file launches Rviz for visualization, and the launch file `bringup_launch.py`, provided by Nav2.

Either of the two parameter files provided should be included in the launch description for Nav2, depending on the parameters to be used. The `navigation_params_differential.yaml` is similar to the default parameter file in Nav2. Only minor modifications to topic names and frames have been made. Using this parameter file Nav2 assumes the robot to use a differential drive, which is not a problem, however, it does not utilize the robot's full potential with an omnidirectional drive. `navigation_params_omnidirectional.yaml` specifies a higher max y velocity which enables the robot to move sideways and a twirling criteria to avoid the robot spinning towards the goal position. The parameter files are a starting point for tuning parameters, and the files may be altered to improve performance.

Chapter 4.

Achievements and Evaluation

To evaluate the simulated robot, a series of experiments targeted at different parts of the system were conducted. The main components of the robotic application were the drive capabilities (Section 4.1), the sensory input and processing, and the motion planning for the LBR. As no motion planning for the LBR was implemented for the digital twin due to time constraints, the experiments focused on the two other main parts. The sensory input and processing were evaluated in two different applications focusing on SLAM (Section 4.2) and navigation (Section 4.3)

The following sections describe the setup and evaluate the performance in each of the three experiments. Videos and complete datasets of the experiments are described in Appendix B

4.1. Holonomic Drive

As the robot's drive in Isaac Sim relies on physically accurate mecanum wheels, rather than overriding the physics engine with a planar mover plugin as in Gazebo, it is necessary to evaluate how the robot responds to command velocities. To assess the accuracy and repeatability of the robot's movement, four scripts found in the kmr_experiments [32] packages were developed.

The four scripts each defined a node publishing twist messages on the `/cmd_vel` topic in a predefined manner. All the experiments were also conducted in reverse to double the data points. The *drive_square* experiment tested the robot's ability to drive in straight lines along the x and y-axis of the robot. The robot did not rotate during this experiment, so movement in the y direction tested the holonomic movement of the robot.

The *drive_diagonal_square* experiment tested the robots holonomic capabilities driving diagonally in all four directions. Again with no rotation. The robot drove in a circle with a velocity in the x-direction and a slight rotation around the z-axis in the *drive_circle* experiment, while in the *drive_rotate* experiment, the robot rotated around its axis with no linear movement. The command velocities sent to the robot in each experiment are presented in Table 4.1. For the drive_square and drive_diagonal_square experiments, a new set of command velocities were published every 10 seconds, which means that it follows the first row of velocity commands the first 10 seconds, the second row the next 10 seconds, and so on until the starting point is reached.

Table 4.1.: Command velocities sent to the robot in the eight experiments conducted for the holonomic drive.

Experiment	**Linear veolcity**			**Angular veolcity**		
	x	y	z	x	y	z
Drive_square	1.0	0.0	0.0	0.0	0.0	0.0
	0.0	1.0	0.0	0.0	0.0	0.0
	-1.0	0.0	0.0	0.0	0.0	0.0
	0.0	-1.0	0.0	0.0	0.0	0.0
Drive_square_reversed	-1.0	0.0	0.0	0.0	0.0	0.0
	0.0	-1.0	0.0	0.0	0.0	0.0
	1.0	0.0	0.0	0.0	0.0	0.0
	0.0	1.0	0.0	0.0	0.0	0.0
Drive_diagonal_square	0.5	0.5	0.0	0.0	0.0	0.0
	-0.5	0.5	0.0	0.0	0.0	0.0
	0.5	-0.5	0.0	0.0	0.0	0.0
	-0.5	-0.5	0.0	0.0	0.0	0.0
Drive_diagonal_square_reversed	-0.5	-0.5	0.0	0.0	0.0	0.0
	0.5	-0.5	0.0	0.0	0.0	0.0
	0.5	0.5	0.0	0.0	0.0	0.0
	-0.5	0.5	0.0	0.0	0.0	0.0
Drive_circle	1.0	0.0	0.0	0.0	0.0	0.25
Drive_circle_reversed	-1.0	0.0	0.0	0.0	0.0	-0.25
Drive_rotate	-1.0	0.0	0.0	0.0	0.0	0.51
Drive_rotate_reversed	-1.0	0.0	0.0	0.0	0.0	-0.51

For each experiment, the path of the robot was tracked, and the following figures (Figure 4.1, 4.2, 4.3) show the results of the drive tests.

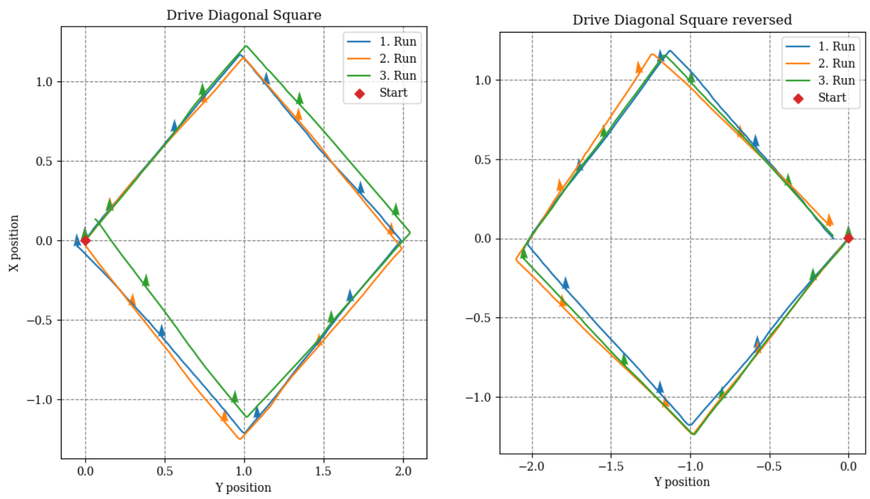


Figure 4.1.: The arrows show the pose of the robots. The robot drives clockwise in both figures but with different starting points. Only linear velocities were applied, and no rotation.

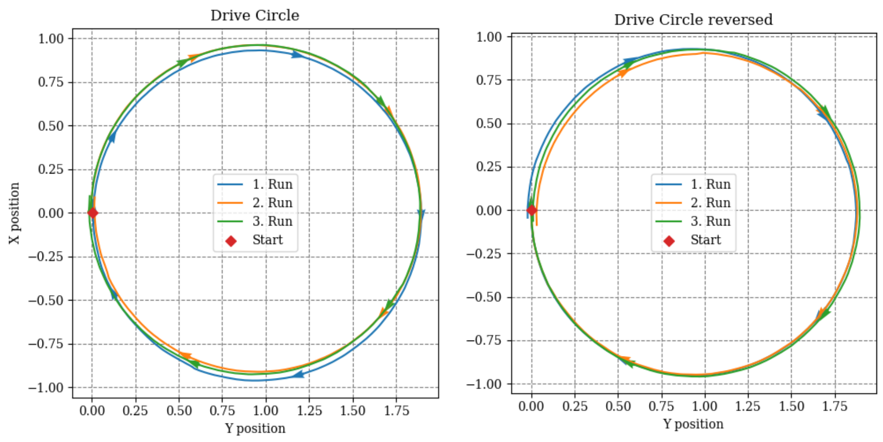


Figure 4.2.: The arrows show the pose of the robots. The left figure drives clockwise, and the right drives counterclockwise. A combination of linear and rotational velocities was applied.

In general, it is clear that the robot follows the path it is expected to, however, for all the experiments, the robot drifts unpredictably. If the robot followed the same path in all experiments, all three lines would perfectly overlap, which is not

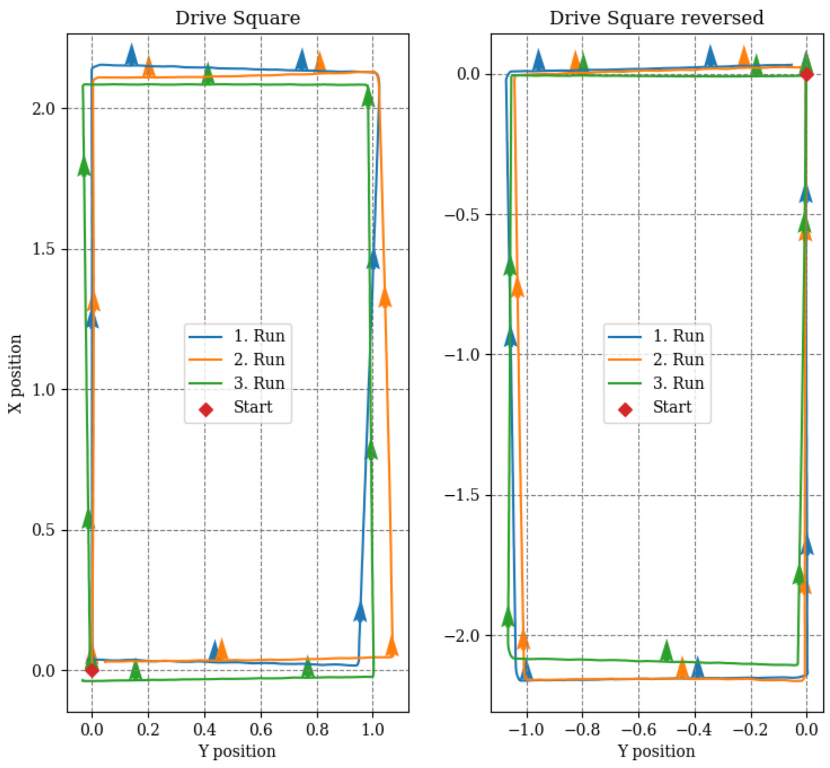


Figure 4.3.: The arrows show the pose of the robots. The robot drives clockwise in both figures but with different starting points. Only linear velocities were applied, and no rotation.

the case. In all the experiments, there is no pattern of how the robot drifts. In neither experiment does the robot perform significantly better than the other.

As mentioned in Subsection 3.1.5, this was, however, a concern when modeling the holonomic drive of the robot. No technical data exists for how the KMR drifts when driving, but there are reasons to assume that the drift is less than in the simulation. Part of the reason for this is that the robot sometimes appears to hit unevennesses in the floor and jump. These minor jumps are visualized in Figure 4.4, which plots the z position of the robot during each of the experiments.

As seen in Figure 4.4, the amount of jumping depends on the type of driving, and it occurs irregularly and, in some cases, not at all (green line in drive_square_reversed). The small vibrations seen in the *drive_square* experiments probably occur due to the shape of the mecanum wheel and the fact that they are not perfectly circular. From the figure, it is clear that driving straight along the x or y direction, as in the

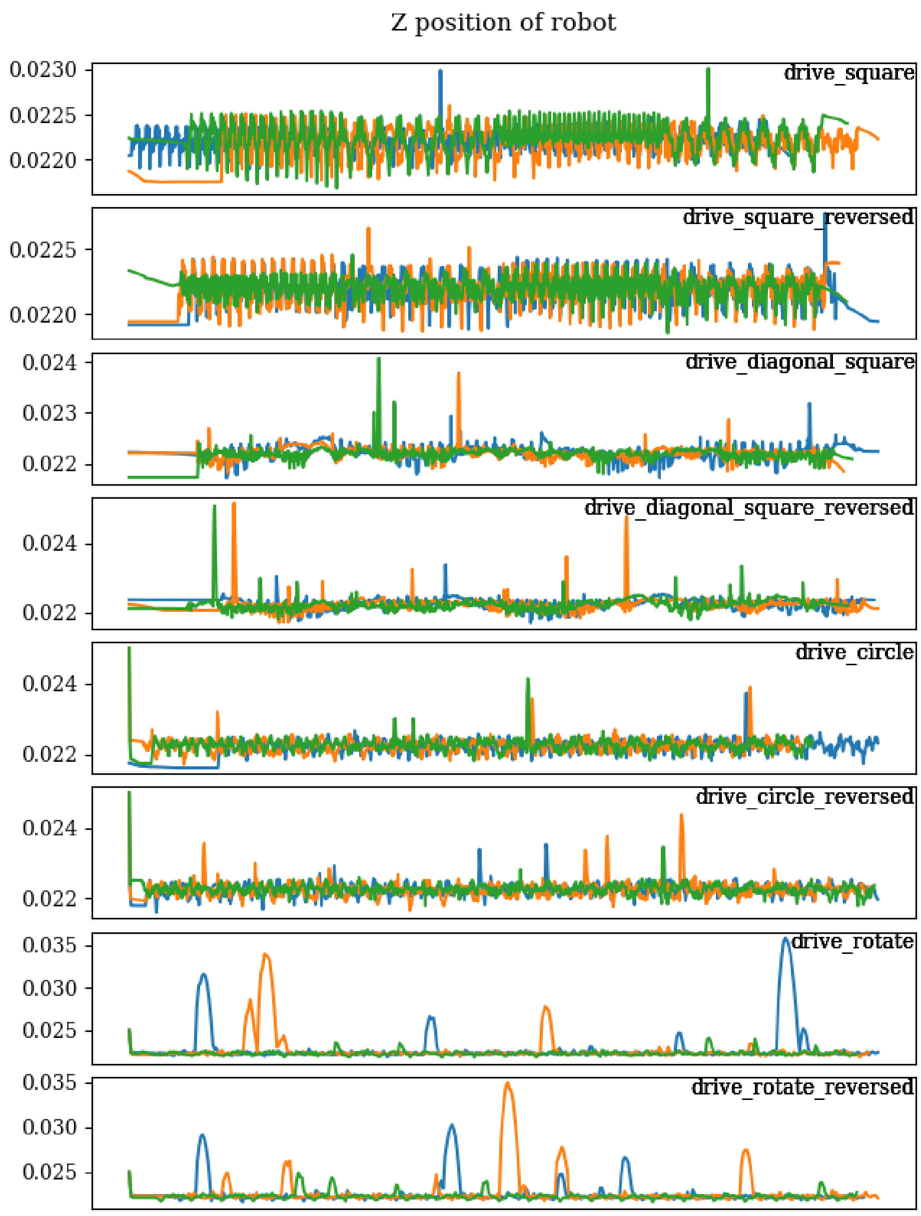


Figure 4.4.: Z position of the robot during the drive experiments. The x-axis is time.

drive_square experiments, causes the least amount of motion in the z-direction. However, driving diagonally and in circles appears to produce approximately the same amount of motion, somewhat more than driving straight. It is clear that the issue is most significant with pure rotations. The plots show that the values are significantly higher than in any of the other experiments.

It should be noted that the robot's speed does not correspond directly to the command velocities sent as twist messages. In the experiment, new command velocities were sent every 10 seconds, which means that for the *drive_square* experiment, the actual velocity was approximately $2.1m/10s = 0.21m/s$ when a command velocity of $1m/s$ in x-direction was intended. It should be possible to change the parameters for the holonomic controller to address this issue, however, the lower speed contributes to less jumping.

To conclude the drive experiments, it can be argued that the main problem of the holonomic drive in the simulator is the jumping which, to some extent, causes drifting. There might be other reasons for the drifting as well, however, the drifting itself is not a critical issue. As the robot is meant to navigate autonomously, this should correct for drifting, as it is a common problem in robotic applications. To prevent jumping, a solution can be to implement suspension for the wheels. In the simulations, the joints are infinitely stiff, which is likely to cause the robot to jump. The jumping may affect the payload placed atop the KMP, and it is a more severe problem that should be fixed. For these experiments, the velocity was lowered to minimize the jumping.

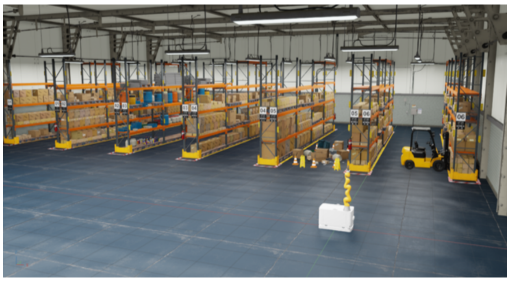
4.2. SLAM

To evaluate the robot's total capabilities, two SLAM experiments were conducted. The aim was for the robot to map two different warehouses and produce an occupancy grid map of each environment. The generated map was compared to the ground truth map generated using the occupancy grid extension included in Isaac Sim [44] to evaluate the results. The robot was manually driven in both experiments using the Teleop twist keyboard package in ROS [13]. The SLAM experiments were conducted with the SLAM configuration presented in subsubsection 3.3.3.

The KMR was tested in the two warehouses shown in Figure 4.5. The warehouses are called *Warehouse with forklifts* and *Full warehouse*.

(a) Warehouse with forklifts

(b) Warehouse with forklifts


(c) Full warehouse

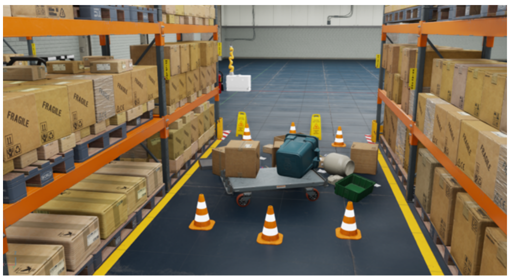
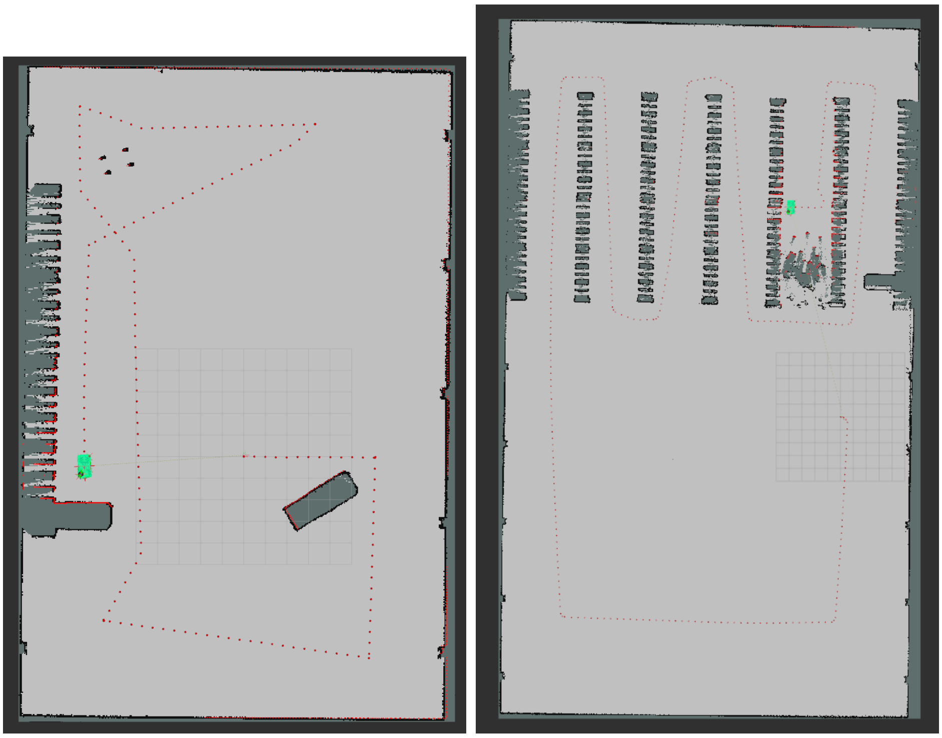

(d) Obstacles in Full warehouse

Figure 4.5.: Pictures from the two simulation environments the robot's SLAM system was tested in.

Figure 4.6 shows the Rviz interface after finished mapping. The figure includes the map and the path the robot drove.

To evaluate the maps, they are compared to the ground truth generated by Isaac sim. The occupancy grid extension [44] generates a map based on all the obstacles in the scene. For this exact map, the lower bound parameter was set to 0.09 m, and the max height was set to 0.1 m. This was done to match the properties of the laser scanners on the KMR. These are not exactly the same values as the



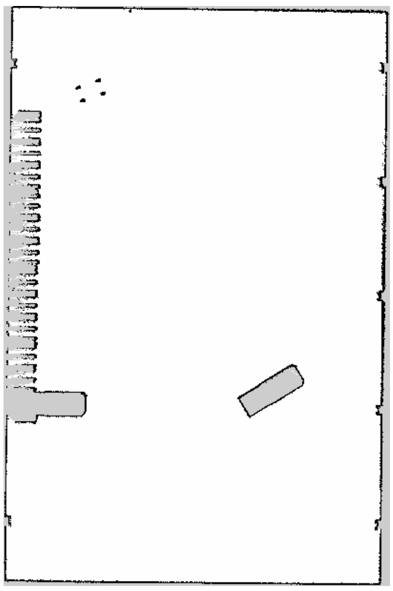
(a) Path in the warehouse with forklifts. (b) Path in the full warehouse.

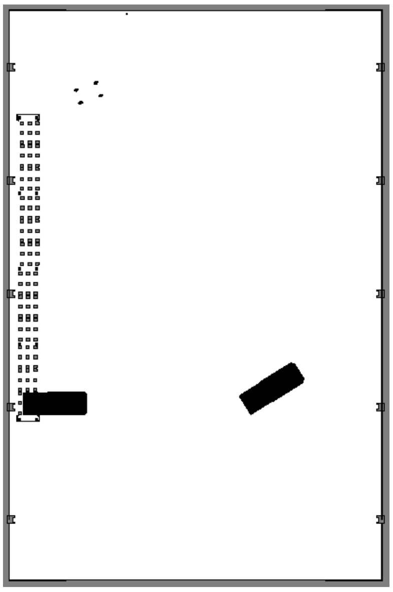
Figure 4.6.: The dotted red line shows the path of the robot when executing the SLAM experiments. The starting point is in the middle of the grid in both figures.

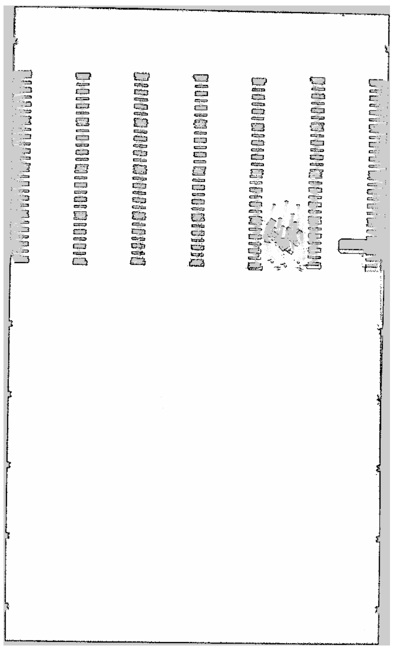
technical specifications in [16], however, experimental results showed this yielded the most similar results.

Figure 4.7 shows the two SLAM-generated maps side by side with the ground truth maps, and from this, it is clear that the maps match very well. As explained in Subsection 2.3.2, SLAM algorithms depend on the odometry frame, `odom`. Isaac Sim publishes this frame from an OmniGraph node from the ROS 2 bridge extension, however, it is not specified how this odometry is calculated in the documentation. It is assumed that it is not calculated directly from the wheel encoders, which would be the case with the real robot, as the `odom` frame drifts little. However, the drift is not zero, which means that some uncertainty is introduced into the system, as is expected from a physical system. Further experiments have to be conducted to evaluate how close the simulation is to reality with regard to odometry.

There are no apparent flaws with the accuracy of the maps, however, a major


(a) SLAM generated map for Warehouse with forklifts


(b) Ground truth map for Warehouse with forklifts


(c) SLAM generated map for Full warehouse

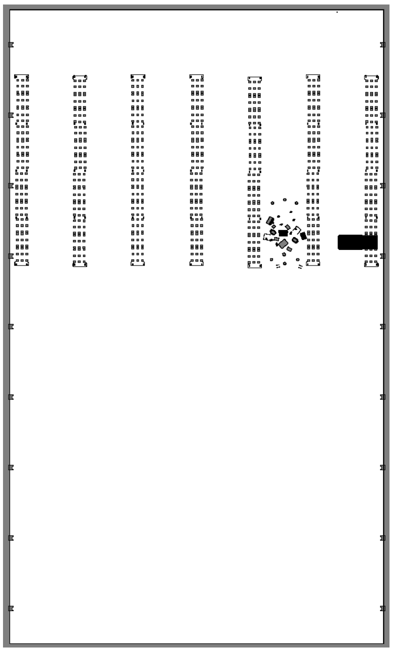

(d) Ground truth map for Full warehouse

Figure 4.7.: The left maps are SLAM-generated maps from the robot while the maps on the right are the ground truth maps for each warehouse, respectively.

issue is that the robot does not detect shelves unless there are pallets underneath it. In all scans, the pallets look like multiple stripes laid side by side. This is the major limitation of the SLAM capabilities of the robot, as it only uses laser scanners.

One possible solution to this problem is to implement *Visual SLAM*, or *vSLAM*. As the robot is equipped with multiple cameras, the images and depth information from these can be used to feed the SLAM algorithm more information and thus create better maps. vSLAM also provides the possibility of generating 3D maps of the environment the robot is operating in. In future development, implementing vSLAM should be of high priority as it will allow the robot to create more accurate maps and thus improve navigation capabilities. Heggem & Whal [10] also concluded that implementing vSLAM significantly increased performance based on their experiments with the physical robot.

There exist different ROS packages and systems developed for vSLAM. NVIDIA also released a developer preview of a hardware-accelerated visual SLAM system [53] tailored to Isaac Sim and NVIDIA hardware that can be implemented and tested in further development of the KMR.

4.3. Navigation

Lastly, an experiment to evaluate the robot's autonomous navigation was conducted. The robot was provided with the map created in the SLAM experiment for the warehouse with forklifts, and two different sets of parameters were used. In both cases, the robot started at its starting position and drove autonomously around the forklift in the middle of the warehouse. Figure 4.8 illustrates the planned path.

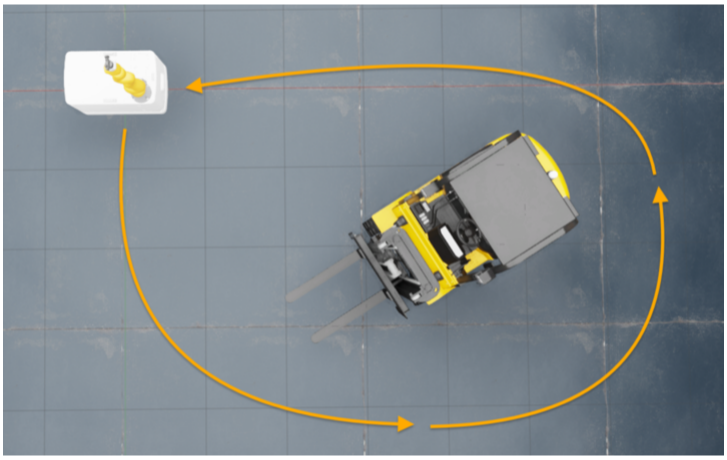


Figure 4.8.: A top-down view of the planned path for the navigation task. Arrows are to illustrate the path and are not part of the simulation

The two experiments were conducted using the same path, and the experiment aimed to evaluate how the robot performed the navigation tasks using different parameters in the navigation file. The main difference was that the first experiment used the default parameters for Nav2, which includes planning for a differential drive robot, whereas the second experiment tested parameters to plan for an omnidirectional robot. In addition, the waypoints around the forklift were set in advance for experiment 1, contrary to experiment 2, where the waypoints were set during execution. The differences are summarized in Table 4.2

Table 4.2.: Summary of differences in navigation experiment 1 and 2.

	Experiment 1	**Experiment 2**
Drive	Differential	Omnidirectional
Waypoints	Set before navigation execution begins	Set during execution

The robot was given its current pose at the start of the task, as required by the Nav2 stack. In addition, three waypoints to follow around the forklift were provided. Figure 4.9 shows the Rviz interface when the execution of the task was begun.

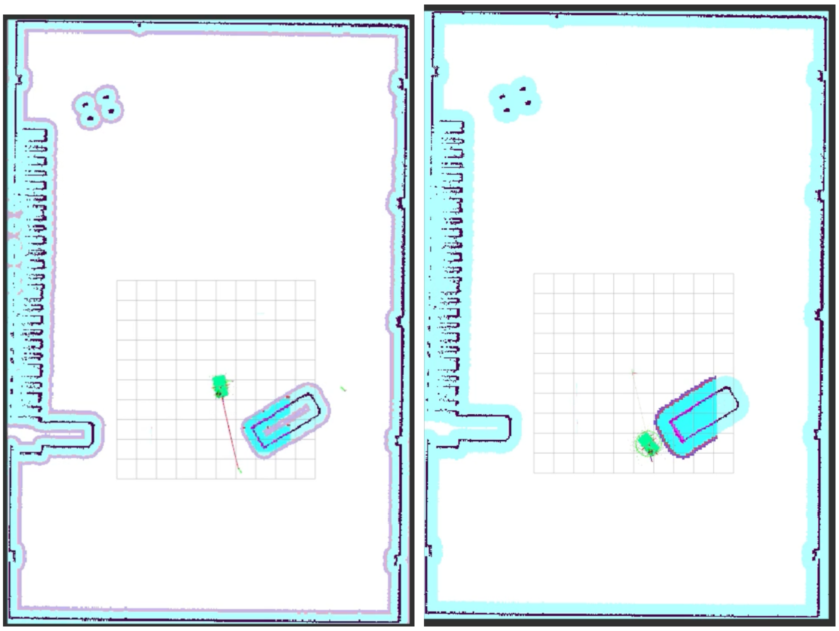


Figure 4.9.: The figure shows the navigation task underway. The left figure with differential drive, and the right figure with omnidirectional drive.

The two experiments used different configuration files to allow Nav2 to plan for a differential or omnidirectional robot. During the experiments, it became apparent that the different parameters significantly impacted the planning. Using the default parameters for differential drive robots was the starting point for tuning each of the two parameter files, however, it turned out that tuning the parameters for the omnidirectional navigation was significantly more complex. Figure 4.10 shows a closeup of the robot navigating around the forklift. In this figure, it is clear that Nav2 plans a path significantly closer to the forklift using the omnidirectional parameters.

To specify the perimeter of the robot Nav2 by default, uses a circle surrounding the robot. This perimeter is illustrated in Figure 4.11a. This is, however, a simplification of the perimeter that limits planning options in tight spaces, as Nav2 assumes the KMR to be wider than it actually is.

During the experiment evaluating the omnidirectional drive, the robot planned a path close to the forklift, and eventually, the radial perimeter overlapped with the mapped forklift as shown in Figure 4.11b. The assumed crash was due to the simplified perimeter, while in reality, the robot did not collide with the forklift.

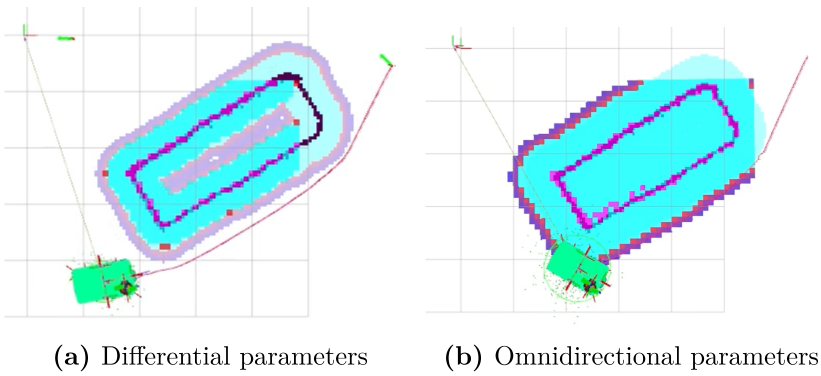

(a) Differential parameters (b) Omnidirectional parameters

Figure 4.10.: The figures show the planned path with the different parameter files.

As recovery behavior was not disabled, the robot performed small movements to escape the collision and replan a local path to reach its goal position. It is not clear why Nav2 planned a path that caused the collision with the forklift. This did not happen when Nav2 planned the path as a differential drive robot.

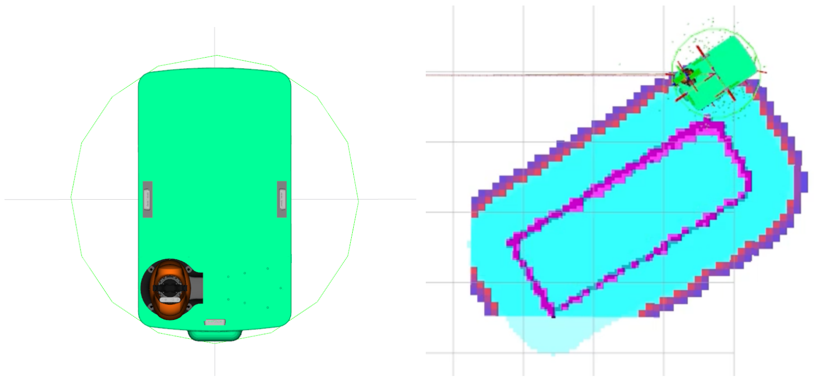

(a) Green circle illustrates the radius parameter (b) Collision detected as circumference overlaps with occupied cell

Figure 4.11.: Figures show when the robot thinks it has collided.

Figure 4.12 shows the executed path followed in both experiments. From this, it is clear that the robot followed a smoother path in the differential experiment, while in the omnidirectional experiment, the robot used a more uneven path. This unevenness indicates that new plans were often re-generated as the previously planned path proved not feasible. In addition to poorly tuned parameters for Nav2, another possible explanation is that the robot does not follow the given velocity commands as expected. As discussed in Section 4.1, the robot does not follow the command velocities exactly though the general direction is usually sufficient. This might cause issues for the local and global planner in Nav2 as the robot does not move exactly as expected. Further work has to be conducted to fix this issue.

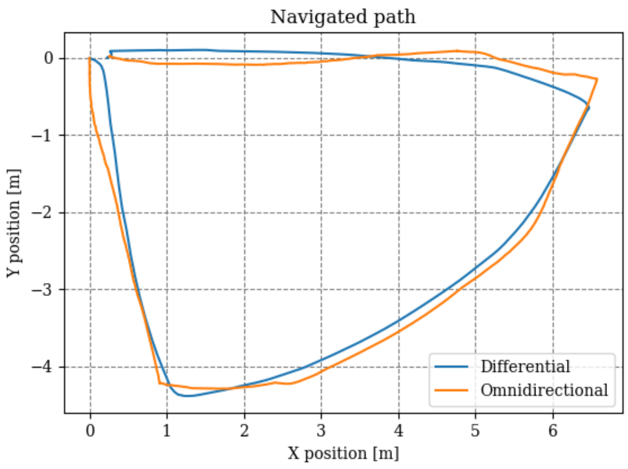


Figure 4.12.: The navigated path. Started in the top left corner and drives counterclockwise

For a properly functioning navigation stack using Nav2, more experimentation and tuning of parameters have to be done. In particular, parameters related to the omnidirectional drive, such as y-velocities and *twirling* parameters. The twirling parameter is used by the DWB controller to prevent holonomic robots from spinning as they make their way to the goal [57]. This parameter proved challenging to tune correctly.

In general, the navigation stack implemented in this thesis works to some extent. It proves that the robot is able to utilize its sensors and actuators, however, for an optimal implementation, more work has to be conducted. As discussed, there are some suspected issues related to the drive capabilities of the robot as well as the parameters regarding the omnidirectional drive. Nevertheless, if the navigation parameters for differential drive robots are used, the navigation stack implemented in this thesis performs satisfactorily.

Chapter 5.

Conclusion and Further Work

As discussed in Chapter 4, there are parts of the system that needs improvements while others work satisfactorily. Section 5.1 describes aspects of the system that needs improvements to function as a viable option for a DT and how the system can be further developed to enhance functionality. Lastly, Section 5.2 concludes the work in this thesis.

5.1. Further Work

This section is divided into three main parts focusing on parts that need improvement (Subsection 5.1.1), suggestions for improvements (Subsection 5.1.2), and lastly, ideas for developing the project further and creating a feature-rich environment for robotic development using Isaac Sim (Subsection 5.1.3).

5.1.1. Requirements

The experiments described in Chapter 4 identified three major parts of the system, which were evaluated; the driving capabilities of the robot, its mapping capabilities, and the ability to navigate an environment autonomously. In addition to the experiment, a final part of the system that should be further developed and evaluated is the manipulator and gripper.

A challenge working with a simulator not allowing developers to bypass the physics engine has its drawbacks as this sets higher requirements for the physical accuracy of the modeled robot. Creating the model was the most time-consuming part of this project, and simplifications and assumptions were made to create a functioning robot. As the project stands, the main issue with the model is the wheels.

At lower speeds, the robot behaves as expected. However, problems occur at higher velocities with the robot jumping, as discussed in Section 4.1. For a functioning DT, this problem has to be fixed. The most likely solution is to integrate suspension in the wheels to model the actual robot and not use simulated infinitely stiff joints.

In addition, the speeds have to be adjusted to match the command velocities given to the robot. For this issue, two possible solutions are proposed. Either the parameters of the HolonomicController have to be tuned to closer match the robot's properties, or the gain parameter of the ArticulationController can be adjusted to match the actual speed through experiments. The former is likely the most robust solution, while the latter is more easily achieved.

Regarding the navigation experiments, the robot performed sufficiently with the differential drive parameters but not utilizing the holonomic drive. The fact that the robot successfully navigated the environment showed that the system developed has potential even though improvements should be made. To improve the navigation capabilities of the robot, further tuning of the parameters should be carried out. The issues discovered here might also relate to the problems driving the robot. Extensive tuning of navigation parameters is time-consuming; however, it is necessary for the robot to navigate autonomously.

The last shortfall of the model is that no motion planning for ROS was implemented for the LBR. Due to time constraints, the integration of the LBR was not prioritized in favor of implementing autonomous navigation and mapping. However, the interface from ROS to the simulator is complete, and simple tests manually passing joint commands over the ROS system prove that the simulator responds as expected. For a viable DT, motion planning should be implemented.

With the latest release of ROS 2 Humble, the MoveIt Setup Assistant was ported from ROS 1, which enables a simpler setup of MoveIt for the LBR. As the latest release of Isaac Sim have beta supports ROS 2 Humble, a possible solution is to upgrade the project to the latest ROS and Isaac Sim versions and then implement motion planning for the manipulator and gripper.

5.1.2. Suggestions

Currently, the SLAM system only relies on two laser scanners to produce a map. Even though it can map vast areas, it has the drawback of only detecting objects approximately 15 cm above the floor. For a more versatile system, the implementation of vSLAM is recommended. This provides the possibility of creating 3D maps and utilizing multiple sensors for higher resolution. Prebuilt ROS packages exist for this purpose, such as OpenVSLAM, which is part of the Nav2 system [30].

In addition, NVIDIA provides a hardware-accelerated ROS package for VSLAM for ROS 2 Humble, which shows promise [53].

To take full advantage of the cameras in the simulator, it is suggested that they are modeled as the Intel RealSense cameras rather than cameras publishing separate RGB and depth images. Isaac Sim provides the possibility of modeling cameras as stereo pairs and in combination with Intel's RealSense ROS packages [48], the simulation is likely to be significantly closer to reality.

Lastly, a common interface for the physical and the simulation robot should be tested. The interface developed for the simulator is based on [10] and is intended to be easily transferrable; however, this is yet to be tested. Therefore, it is suggested that the two systems are combined to provide a virtual testbed for efficiently developing robotic applications.

5.1.3. Going Further

The system presented in this thesis is designed to be modular and easily extendable, and there are multiple approaches to further advance the system. Integrating computer vision pipelines for object detection or other computer vision tasks will allow the robot to perform a broader set of tasks regarding navigation and mapping and pose estimation for grasping. This leverages Isaac Sim's photorealistic capabilities to develop, test, and optimize such pipelines, enabling more versatile robots. Isaac Replicator [44] is a tool in the NVIDIA ecosystem specifically designed for this purpose which provides smother integration. Transferring such computer vision pipelines onto the physical robot will likely require tuning. However, synthetic data generation in computer vision pipelines has previously been utilized successfully.

Isaac Sim provides *hardware in the loop* features that allow real-time operations of the physical robot through the simulator, using ROS. If such a system were to be integrated, this would provide a significantly more flexible working environment for the robot utilizing Isaac Sim extensions and ROS. Isaac Cortex [44] is a decision-making framework, still only in preview release, which provides simple examples of utilizing hardware in the loop with ROS.

As of now, the KMR has only been tested in prebuilt environments. However, for a more accurate digital twin, a model of the MANULAB at NTNU could be imported into the simulator. There have been examples [11] of robots being trained to navigate the campus of ETH Zürich in a laser-scanned model of the environment. This could be an interesting approach to further develop the system to more accurately test and optimize the system.

In Industry 4.0, multiple robots will collaborate in the same environment, and with two KMRs present at the MANULAB multi-robot, collaboration would be an interesting enhancement to the system. Such collaboration can include moving objects too heavy or large for an individual robot or assembling components requiring two manipulators. For this development, using a digital twin will be beneficial as this allows for rapid prototyping and optimization without risking damaging robots or human personnel.

5.2. Concluding Remarks

This thesis aimed to create a digital twin of the KUKA KMR iiwa utilizing NVIDIA Isaac Sim to streamline the development, testing, optimization, and deployment of robotic applications. In addition, the thesis presents a novel development framework for the KMR based on ROS and Isaac Sim, which simplifies the development of robotic applications.

A physically accurate model of the KMR has been developed to use in the simulated environment to replicate the actual robot as closely as possible. Considering very few resources for modeling the robot are provided by the manufacturer, the final model of the robot is close enough to the actual robot that it is regarded as successful.

Experiments have been conducted validating the sensors and actuators in the simulated environment, and a SLAM and Navigation system has been implemented. In simulations, the SLAM system works as expected, generating accurate maps close to the ground truth. Furthermore, the system utilized the laser scanners with no issues, and the integration is considered successful.

A navigation system was implemented with some success. Two different navigation configurations have been developed with varying degrees of success. Using the first differential configuration, the robot navigates its environment with few issues and performs satisfactorily, while the omnidirectional configuration reviles flaws in the system. Overall the navigation system proved that it has potential and is close to a fully functioning navigation system.

Due to time-consuming issues related to modeling the robot accurately enough to be of value, the system has not yet been configured with the physical robot. Further development will have to address this for the system to work as a digital twin. However, the system has been designed with the physical robot as an intended part of the final system, likely simplifying integration.

As an initial effort to create a digital twin in a newly released simulation environment, it can be argued that the results are a success as multiple subsystems of the robot have been integrated. Furthermore, the complete configuration, consisting of the ROS system and Isaac Sim extensions, was designed to be easily maintainable and further developed. As a result, the final system shows great potential, and with further development, this can be an essential contribution to developing robotic applications, particularly for the KUKA KMR iiwa.

Appendix A.

Cofiguring ROS 2 with Isaac Sim

Isaac Sim comes with its ROS distribution and might cause issues when running the local version of ROS sourced from `/opt/ros/foxy/setup.bash` simultaneously. One common error is something along the lines of `[RTPS_TRANSPORT_SHM Error] Failed init_port fastrtps_port7412: open_and_lock_file failed ->`
`Function open_port_internal`. This relates to DDS and can be solved using the following steps.

1. If applicable, remove `source /opt/ros/foxy/setup.bash` from the `.bashrc` file and source the ROS installation when a new terminal is opened if needed.
2. Copy the DDS configuration file from Listing A.1 to a location on your computer e.g. ∼`/.ros/fastdds.xml`
3. Add the following two lines to `.bashrc` file

   ```
   export FASTRTPS_DEFAULT_PROFILES_FILE=~/.ros/fastdds.xml
   export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
   ```

The code below is provided by NVIDIA and is the DDS configuration used to sort out the issue.

Listing A.1: DDS configuration file provided by NVIDIA [52]

```
<?xml version="1.0" encoding="UTF-8" ?>

<!--
Copyright (c) 2022, NVIDIA CORPORATION. All rights reserved.

```

```
NVIDIA CORPORATION and its licensors retain all intellectual
    ↪ property
and proprietary rights in and to this software, related
    ↪ documentation
and any modifications thereto. Any use, reproduction, disclosure or
distribution of this software and related documentation without an
    ↪ express
license agreement from NVIDIA CORPORATION is strictly prohibited.
-->

<license>NVIDIA Isaac ROS Software License</license>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/
    ↪ fastRTPS_Profiles" >
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>UdpTransport</transport_id>
            <type>UDPv4</type>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="udp_transport_profile"
        ↪ is_default_profile="true">
        <rtps>
            <userTransports>
                <transport_id>UdpTransport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
        </rtps>
    </participant>
</profiles>
```

Appendix B.

Datasets and Videos

B.1. Datasets

The dataset on which all the plots are based is located in the omniverse-extension repository [33] under the data_processing/data directory. A readme in the repository briefly explains the dataset.

B.2. Videos

The following videos are screen recordings of the experiments related to SLAM and navigation.

- SLAM Experiment - Warehouse with Forklifts:
  https://youtu.be/DVSTiLKzIws
- SLAM Experiment - Full Warehouse:
  https://youtu.be/jBqepVnKZe8
- Navigation experiment 1 - Differential Drive, Warehouse with forklifts:
  https://youtu.be/7BlMSwbD0-w
- Navigation experiment 2 - Omnidirectional drive, Warehouse with Forklifts:
  https://youtu.be/gl8n8-Ab5j0

References

[1] Mahmoud Abdelrahim, Amgad Hassan, Damilare Ojo, Mona Hosny, Hossam Hassan Ammar, and Mahmoud El-Samanty. “A Novel Design of a T-Model Three Mecanum Wheeled Mobile Robot”. In: *2022 International Conference on Advanced Robotics and Mechatronics (ICARM)*. 2022, pp. 509–513. DOI: 10.1109/ICARM54641.2022.9959189.

[2] Paul Bovbel, Michel Hidalgo, and Tully Foote. *pointcloud_to_laserscan*. 2022. URL: https://index.ros.org/p/pointcloud_to_laserscan/ (visited on 01/03/2023).

[3] Luca Carlone, Rosario Aragues, José A. Castellanos, and Basilio Bona. “A fast and accurate approximation for planar pose graph optimization”. In: *The International Journal of Robotics Research* 33.7 (2014), pp. 965–987. DOI: 10.1177/0278364914523689.

[4] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. “A Review of Physics Simulators for Robotic Applications”. In: *IEEE Access* 9 (2021), pp. 51416–51431. DOI: 10.1109/ACCESS.2021.3068769.

[5] Morten M. Dahl. *Configuration, Calibration and Control of Mobile Manipulators for Industry 4.0*. Dec. 2020.

[6] Tully Foote and Mike Purvis. *Standard Units of Measure and Coordinate Conventions*. 2010. URL: https://www.ros.org/reps/rep-0103.html (visited on 12/22/2022).

[7] Brian Gerkey and Tony Pratkanis. *map_server*. 2020. URL: http://wiki.ros.org/map_server (visited on 05/02/2022).

[8] Robert Haschke. *xacro*. 2021. URL: https://github.com/ros/xacro/wiki (visited on 12/22/2022).

[9] Charlotte Heggem and Nina Marie Wahl. *Configuration and Control of KMR iiwa with ROS2*. Dec. 2019.

[10] Charlotte Heggem and Nina Marie Wahl. “Mobile Navigation and Manipulation: Configuration and Control of the KMR iiwa with ROS2”. MA thesis. Norwegian University of Technology and Science, June 2020. URL: https://hdl.handle.net/11250/2781684.

[11] David Hoeller. *Learning Challenging Tasks For Quadrupedal Robots: From Simulation To Reality.* Nov. 2021. URL: https://www.nvidia.com/en-us/on-demand/session/gtcfall21-a31308/ (visited on 11/26/2022).

[12] Intel. *Intel® RealSense™ Depth Camera D435.* 2022. URL: https://www.intelrealsense.com/depth-camera-d435/ (visited on 12/04/2022).

[13] Graylin Trevor Jay. *teleop_twist_keyboard.* 2022. URL: https://github.com/ros2/teleop_twist_keyboard (visited on 01/10/2023).

[14] Sören Kerner and Julian Eßer. *Towards a Digital Reality in Logistics Automation: Optimization of Sim-to-Real.* Mar. 2022. URL: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s42559/ (visited on 11/26/2022).

[15] Stefan Kohlbrecher and Johannes Meyer. *hector_gazebo.* 2013. URL: https://wiki.ros.org/hector_gazebo (visited on 12/04/2022).

[16] KUKA. *Mobile Robots KMP 200 omniMove Transport Vehicle Assembly and Operating Instructions.* Requires login as KUKA customer. 2022. URL: https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB9937_en (visited on 01/10/2023).

[17] KUKA. *Mobile Robots KMR iiwa omniMove Mobile Industrial Robot System Assembly and Operating Instructions.* Requires login as KUKA customer. 2022. URL: https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB9761_en (visited on 01/13/2023).

[18] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. "G2o: A general framework for graph optimization". In: *2011 IEEE International Conference on Robotics and Automation.* 2011, pp. 3607–3613. DOI: 10.1109/ICRA.2011.5979949.

[19] Mathieu Labbé and François Michaud. "Long-term online multi-session graph-based SPLAM with memory management". In: *Autonomous Robots* 42.6 (Aug. 2018), pp. 1133–1150. ISSN: 1573-7527. DOI: 10.1007/s10514-017-9682-5.

[20] Chris Lalancette. *robot_state_publisher.* 2022. URL: https://github.com/ros/robot_state_publisher (visited on 01/03/2023).

[21] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. "Industry 4.0". In: *Business & Information Systems Engineering* 6.4 (Aug. 2014), pp. 239–242. ISSN: 1867-0202. DOI: 10.1007/s12599-014-0334-4.

[22] David V. Lu. *Fundamentals of Local Planning.* Unpublished manuscript, Conference notes. 2017. URL: https://roscon.ros.org/2017/presentations/ROSCon%5C%202017%5C%20Fundamentals%5C%20of%5C%20Local%5C%20Planning.pdf.

[23] David V. Lu and Jackie Kay. *joint_state_publisher.* 2022. URL: https://index.ros.org/p/joint_state_publisher/ (visited on 01/03/2023).

[24] David V. Lu and Jackie Kay. *joint_state_publisher_gui.* 2022. URL: https://index.ros.org/p/joint_state_publisher_gui/ (visited on 01/03/2023).

[25] Kevin M. Lynch and Frank C. Park. *Modern Robotics.* Cambridge, England: Cambridge University Press, 2017. ISBN: 978-1-10-715630-2.

[26] Steve Macenski and Ivona Jambrecic. "SLAM Toolbox: SLAM for the dynamic world". In: (May 2021). DOI: 10.21105/joss.02783. URL: https://joss.theoj.org/papers/10.21105/joss.02783.

[27] Steve Macenski, Francisco Martin, Ruffin White, and Jonatan Gines Clavero. "The Marathon 2: A Navigation System". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, Oct. 2020. DOI: 10.1109/iros45743.2020.9341207. URL: https://doi.org/10.1109%5C%2Firos45743.2020.9341207.

[28] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[29] Wim Meeussen. *Coordinate Frames for Mobile Platforms.* 2010. URL: https://www.ros.org/reps/rep-0105.html (visited on 12/22/2022).

[30] Alexey Merzlyakov and Steve Macenski. "A Comparison of Modern General-Purpose Visual SLAM Approaches". In: (2021). DOI: 10.48550/ARXIV.2107.07589. URL: https://arxiv.org/abs/2107.07589.

[31] Jørgen Usterud Myrvold. *Digital Twins of Mobile Manipulators: Focused on KUKA KMR iiwa.* Added as attachment. May 2022.

[32] Jørgen Usterud Myrvold. *kmr_ws.* 2023. URL: https://github.com/jorgenmyrvold/kmr_ws (visited on 01/04/2023).

[33] Jørgen Usterud Myrvold. *omniverse-extensions.* 2023. URL: https://github.com/jorgenmyrvold/omniverse-extensions (visited on 01/07/2023).

[34] NVIDIA. *Discover Kit SDK.* 2022. URL: https://docs.omniverse.nvidia.com/prod_kit/prod_kit/overview.html (visited on 12/29/2022).

[35] NVIDIA. *Fluid Dynamics UAV, blimp, subsea*. 2021. URL: https://forums.developer.nvidia.com/t/fluid-dynamics-uav-blimp-subsea/187145/2?u=user164 (visited on 04/20/2022).

[36] NVIDIA. *Isaac SDK*. 2022. URL: https://docs.nvidia.com/isaac/isaac/doc/index.html (visited on 04/20/2022).

[37] NVIDIA. *Isaac Sim*. 2023. URL: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/isaac-sim (visited on 01/12/2022).

[38] NVIDIA. *Isaac Sim: Extension API*. 2022. URL: https://docs.omniverse.nvidia.com/py/isaacsim/index.html (visited on 01/03/2023).

[39] NVIDIA. *Multi robot control support*. 2021. URL: https://forums.developer.nvidia.com/t/multi-robot-control-support/184937/2?u=user164 (visited on 04/20/2022).

[40] NVIDIA. *NVIDIA Isaac ROS*. 2022. URL: https://github.com/NVIDIA-ISAAC-ROS (visited on 01/04/2022).

[41] NVIDIA. *NVIDIA Isaac Sim*. 2022. URL: https://developer.nvidia.com/isaac-sim (visited on 11/26/2022).

[42] NVIDIA. *NVIDIA Omniverse*. 2022. URL: https://www.nvidia.com/en-us/omniverse/ (visited on 11/26/2022).

[43] NVIDIA. *Omniverse Nucleus*. 2022. URL: https://docs.omniverse.nvidia.com/prod_nucleus/prod_nucleus/overview.html (visited on 12/29/2022).

[44] NVIDIA. *Omniverse: Isaac Sim*. 2022. URL: https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim.html (visited on 04/20/2022).

[45] NVIDIA. *Physics Core - Omniverse Create*. 2022. URL: https://docs.omniverse.nvidia.com/app_create/prod_extensions/ext_physics.html (visited on 04/20/2022).

[46] Andrew Price. *robotiq_arg85_description*. 2020. URL: https://github.com/a-price/robotiq_arg85_description (visited on 11/29/2022).

[47] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, and TaeHoon Lim. *ROS Robot Programming: A Handbook is written by TurtleBot3 Developers*. ROBOTIS Co.,Ltd., 2017. ISBN: 979-11-962307-1-5.

[48] Intel® RealSense™. *realsense-ros*. 2022. URL: https://github.com/IntelRealSense/realsense-ros (visited on 11/29/2022).

[49] Open Robotics. *ROS 2 Documentation: Foxy*. 2022. URL: https://docs.ros.org/en/foxy/ (visited on 10/27/2022).

[50] Robotiq. *2F-85 and 2F-140 Grippers*. 2022. URL: https://robotiq.com/products/2f85-140-adaptive-robot-gripper?ref=nav_product_new_button (visited on 12/04/2022).

[51] Michael Schluse and Juergen Rossmann. "From simulation to experimentable digital twins: Simulation-based development and operation of complex technical systems". In: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. 2016, pp. 1–6. DOI: 10.1109/SysEng.2016.7753162.

[52] Hemal Shah. *rtps_udp_profile.xml*. 2022. URL: https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_common/blob/main/docker/middleware_profiles/rtps_udp_profile.xml (visited on 10/27/2022).

[53] Hemal Shah and Javieer Singh. *isaac_ros_visual_slam*. 2022. URL: https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_visual_slam (visited on 01/10/2023).

[54] Cyrill Stachniss, John J. Leonard, and Sebastian Thrun. "Springer Handbook of Robotics". In: Berlin, Heidelberg: Springer, 2008. Chap. 46. Simultaneous Localization and Mapping.

[55] stoic-roboticist. *kmriiwa_ros_stack*. 2021. URL: https://github.com/stoic-roboticist/kmriiwa_ros_stack (visited on 09/03/2022).

[56] Pixar Animation Studios. *Universel Scene Description*. 2021. URL: https://graphics.pixar.com/usd/ (visited on 11/26/2022).

[57] Nav2 Team. *Nav2*. 2023. URL: https://navigation.ros.org/index.html (visited on 01/10/2023).

[58] Marvin Wiedemann, Anna Vasileva, and Renato Gasoto. *O3DynSimModel*. Nov. 2022. URL: https://git.openlogisticsfoundation.org/silicon-economy/simulation-model/o3dynsimmodel (visited on 11/26/2022).

[59] Brian Wilcox. *ROS2 Index: nav2_map_server*. 2022. URL: https://index.ros.org/p/nav2_map_server/ (visited on 05/02/2022).

[60] Hirotaka Yamada, Steve Peters, and Ilya Pankov. *urdf/XML Specification*. 2022. URL: http://wiki.ros.org/urdf/XML (visited on 12/22/2022).

[61] Iuliu Zamfirescu and Carlos Pascal. "Modelling and simulation of an omnidirectional mobile platform with robotic arm in CoppeliaSim". In: *2020 24th International Conference on System Theory, Control and Computing (ICSTCC)*. 2020, pp. 667–672. DOI: 10.1109/ICSTCC50638.2020.9259773.