

Charlotte Heggem  
Nina Marie Wahl

# Mobile Navigation and Manipulation

Configuration and Control of the KMR iiwa with  
ROS2

Master's thesis in Engineering & ICT

Supervisor: Lars Tingelstad

June 2020





Charlotte Heggem  
Nina Marie Wahl

# **Mobile Navigation and Manipulation**

Configuration and Control of the KMR iiwa with ROS2

Master's thesis in Engineering & ICT  
Supervisor: Lars Tingelstad  
June 2020

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



# Preface

This thesis completes our master's degree within Engineering & ICT at the Norwegian University of Science and Technology in Trondheim. The project is conducted for the research group Robotics and Automation at the Department of Mechanical and Industrial Engineering. The work was carried out during the spring of 2020.

The work is conducted at the Norwegian Manufacturing Research Laboratory, MANULAB, at NTNU in Trondheim. Due to the Covid-19 situation, the MANULAB was closed for eight weeks, and after reopening, the access was restricted. Due to this, priorities have been required for feasible work.

We would like to thank our supervisor Lars Tingelstad, for guidance, encouragement and support during this project. We would also like to thank Adam Leon Kleppe for technical assistance in the laboratory.

We would also like to thank our friends and family for their support and valuable input and discussions. We are grateful that you are always available for a chat, and have helped us keep our motivation and focus during these long and hard-working days.

This thesis assumes the reader has basic knowledge within robotics, programming and ICT. It is advantageous to be familiar with the robot operating system ROS. We hope this thesis will be beneficial for anyone with interest within KMR iiwa robots and ROS2.

Charlotte Heggem & Nina Marie Wahl  
10-06-2020



# Abstract

This thesis investigates how to operate a mobile robot, the KMR iiwa, manufactured by KUKA, through interaction with ROS2. The KMR iiwa consists of a robot arm, LBR iiwa 14 R820, mounted on the base of a mobile platform, KMP 200 omniMove.

It is desired that the implemented system accomplish a fetch and carry scenario between work stations. The system utilizes the mobile base and the manipulator, the integrated sensors of the KMR iiwa, in addition to external sensors and actuators. To enable transmission of data between the devices and the ROS2 framework, a communication architecture is developed. Different ROS2 stacks are used to perform the subtasks, which are split into SLAM, mobile navigation, motion planning and control of a manipulator and object detection. A behavior tree, which is a decision-making mechanism, connects the system.

A comprehensive system that is suitable for the environment and the tools available at the laboratory is developed. The system is verified through experiments where the robot is interacting with the relevant ROS2 stacks. It was desired to complement the integrated lidars of the KMR iiwa with RGB-D cameras to capture the environment in three-dimensional space. Configurations of sensors and techniques have been investigated to create a map of the environment that fully represents the scene. The ROS2 stack for navigation required comprehensive tuning to interact with a large and rectangular robot with holonomic drive. The results presented within navigation concerns how the KMR iiwa can be operated safely in the environment. Within manipulation, motion commands available for the LBR have been explored to make it compatible with the ROS2 motion planner. A proof-of-concept object detection model is developed to localize objects to be grasped and carried by the robot. Finally, suggestions on how the system can be further developed are presented.

This thesis presents a robot system with high relevance and potential for the industry. The system creates a foundation for being able to perform further research and interesting experiments with the KMR iiwa.



# Sammendrag

Denne masteroppgaven omhandler å kontrollere en mobil robot, KMR iiwa, produsert av KUKA, via interaksjon med ROS2. En KMR iiwa består av en robot arm, LBR iiwa 14 R820, som er montert på en mobil platform, KMP 200 omni-Move.

Det er ønskelig at det implementerte systemet kan gjennomføre et scenario der roboten kan plukke opp og frakte objekter mellom arbeidsstasjoner. Til dette bruker systemet den mobile basen og manipulatorene, i tillegg til eksterne sensorer og aktuatorer. En kommunikasjonsarkitektur er utviklet for å overføre data mellom enhetene og ROS2. Deretter blir ROS2 pakker brukt for å gjennomføre de ulike deloppgavene. Disse kan deles inn i SLAM, navigering, bevegelse av manipulator og gjenkjenning av objekter. Konseptet behavior tree brukes for å ta avgjørelser og bytte mellom oppgaver, noe som knytter hele systemet sammen.

Et helhetlig system som er tilpasset miljøet og det utstyret som er tilgjengelig på robotlaboratoriet har blitt utviklet. Systemet er verifisert gjennom eksperimenter der roboten bruker de ulike ROS2 pakkene til å gjennomføre oppgaver. For å kunne observere hele miljøet var det ønskelig å komplementere de integrerte laserne på KMR iiwaen med RGB-D kameraer. Ulike sensorkonfigurasjoner og teknikker har derfor blitt undersøkt og testet for å lage et kart som representerer det fullstendige miljøet på laboratoriet. Ettersom KMR iiwaen er en stor, rektangulær robot som kjører holonomisk, krevde ROS2 pakken som er brukt for navigering grundig gjennomgang av parametrene. Resultatet er en robot som trygt kan navigere rundt på egenhånd. For å bruke manipulatorene måtte roboten gjøres kompatibel med bevegelsesplanleggeren i ROS2. Dette krevde undersøkelser av de tilgjengelige bevegelsene for LBRen. En enkel detekteringsmodell er brukt til å lokalisere objekter som kan plukkes opp og fraktes på roboten. Til slutt presenterer vi forslag til hvordan systemet kan utvikles videre.

Denne masteroppgaven presenterer et robotsystem som er relevant, og har stort potensial, for industrien. Det implementerte systemet danner et grunnlag for å kunne gjennomføre videre forskning og interessante eksperimenter med en KMR iiwa.





# Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>Sammendrag</b>	<b>v</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Background and Motivation . . . . .	1
1.2. Previous Work . . . . .	2
1.2.1. Challenges . . . . .	3
1.3. Contributions . . . . .	4
1.4. Problem Description . . . . .	4
1.5. Related Work . . . . .	5
1.6. Outline . . . . .	7
 <b>I. Fundamentals</b>	 <b>9</b>
<b>2. Preliminaries</b>	<b>11</b>
2.1. Behavior Trees . . . . .	11
2.1.1. Types of Nodes . . . . .	12
2.2. Computer Vision . . . . .	14
2.2.1. Projective Geometry . . . . .	14
2.2.2. Camera Model . . . . .	15
2.2.3. Camera Calibration . . . . .	17
2.2.4. Stereo Vision . . . . .	17
2.2.5. Image Rectification . . . . .	20
2.2.6. Features . . . . .	20
2.2.7. Segmentation . . . . .	21
2.2.8. Object Detection . . . . .	21
2.3. Simultaneous Localization and Mapping . . . . .	24
2.3.1. Map . . . . .	24
2.3.2. Online and Full SLAM . . . . .	25

2.3.3.	Filter and Optimization SLAM . . . . .	26
2.3.4.	Local and Global SLAM . . . . .	26
2.3.5.	Maximum A Posteriori Estimate . . . . .	28
2.3.6.	Localization . . . . .	29
2.3.7.	Mapping . . . . .	31
<b>3.</b>	<b>Hardware</b>	<b>33</b>
3.1.	KMR iiwa . . . . .	33
3.1.1.	Operating Modes . . . . .	34
3.1.2.	Software . . . . .	35
3.1.3.	KMP 200 omniMove . . . . .	35
3.1.4.	LBR iiwa 14 R820 . . . . .	38
3.2.	Robotiq 2F-85 Gripper . . . . .	44
3.2.1.	Gripper Register Mapping . . . . .	45
3.2.2.	Modbus RTU Communication . . . . .	48
3.3.	Intel Realsense Depth Camera D435 . . . . .	48
3.3.1.	ROS Software . . . . .	50
3.3.2.	Calibration . . . . .	51
<b>4.</b>	<b>ROS2</b>	<b>53</b>
4.1.	Introduction to ROS . . . . .	53
4.2.	Concepts . . . . .	54
4.3.	Stacks for SLAM . . . . .	57
4.3.1.	Cartographer . . . . .	57
4.3.2.	RTAB-Map . . . . .	58
4.4.	Behaviortree.CPP . . . . .	59
4.5.	Navigation2 . . . . .	60
4.6.	MoveIt2 . . . . .	64
4.6.1.	Configurations . . . . .	64
4.6.2.	How MoveIt Works . . . . .	65
4.7.	Stacks for Object Detection . . . . .	67
4.7.1.	OpenVINO Toolkit . . . . .	68
4.7.2.	Object Analytics . . . . .	70
<b>II.</b>	<b>Achievements and Evaluation</b>	<b>73</b>
<b>5.</b>	<b>System Description</b>	<b>75</b>
5.1.	Setup . . . . .	75
5.1.1.	Physical Installations . . . . .	75
5.1.2.	Software Setup . . . . .	78

5.2. Robot Model and Simulation . . . . .	82
5.2.1. URDF . . . . .	82
5.2.2. Gazebo . . . . .	82
5.3. Physical Architecture . . . . .	84
5.4. Communication Architecture . . . . .	85
5.4.1. Remote PC . . . . .	87
5.4.2. Sunrise Cabinet . . . . .	90
5.5. Sunrise Application . . . . .	92
5.6. Behavior Tree . . . . .	94
5.6.1. Tree Nodes . . . . .	96
5.7. Server Nodes . . . . .	98
5.7.1. BehaviorTree Node . . . . .	98
5.7.2. NavigationSupport Node . . . . .	99
5.7.3. RunMoveIt Node . . . . .	100
5.7.4. Gripper Node . . . . .	101
5.7.5. ObjectDetection Node . . . . .	102
<b>6. System Review</b>	<b>105</b>
6.1. Communication Architecture . . . . .	105
6.1.1. Communication Protocol . . . . .	106
6.1.2. Network . . . . .	106
6.2. Sunrise . . . . .	107
6.2.1. KMP Sensor Data . . . . .	107
6.2.2. KMP Motions . . . . .	108
6.2.3. LBR Motions . . . . .	109
6.2.4. Safety . . . . .	109
6.3. ROS . . . . .	111
6.3.1. Actions . . . . .	111
6.3.2. Programming Language . . . . .	112
6.4. Sensors and Actuators . . . . .	112
6.4.1. Robotiq 25-85 Gripper . . . . .	112
6.4.2. Intel Realsense Cameras . . . . .	113
6.5. Remarks . . . . .	114
<b>7. SLAM</b>	<b>115</b>
7.1. Experimental Environment . . . . .	115
7.2. Results . . . . .	116
7.2.1. Odometry . . . . .	117
7.2.2. Mapping . . . . .	119
7.2.3. Final Maps . . . . .	125
7.2.4. Mapping the Robot Cells . . . . .	125

7.3.	Discussion . . . . .	126
7.3.1.	Odometry . . . . .	127
7.3.2.	Mapping . . . . .	128
7.3.3.	Remarks . . . . .	131
<b>8.</b>	<b>Navigation</b>	<b>133</b>
8.1.	Results . . . . .	133
8.1.1.	Velocities . . . . .	133
8.1.2.	Dynamic Replanning . . . . .	136
8.1.3.	Dynamic Adjustment of Velocities . . . . .	138
8.1.4.	Voxel Layer with Camera Data . . . . .	138
8.2.	Discussion . . . . .	140
8.2.1.	Tuning . . . . .	140
8.2.2.	Evaluation of Results . . . . .	145
8.2.3.	Remarks . . . . .	147
<b>9.</b>	<b>Manipulation</b>	<b>149</b>
9.1.	Experimental Setup . . . . .	149
9.2.	Results . . . . .	149
9.2.1.	Joint States . . . . .	149
9.2.2.	Path Planning . . . . .	152
9.3.	Discussion . . . . .	155
9.3.1.	Hand Eye Calibration . . . . .	155
9.3.2.	Joint States . . . . .	156
9.3.3.	Path Planning . . . . .	156
9.3.4.	Remarks . . . . .	158
<b>10.</b>	<b>Object Detection</b>	<b>159</b>
10.1.	Results . . . . .	159
10.1.1.	Training . . . . .	159
10.1.2.	Experiments . . . . .	160
10.2.	Discussion . . . . .	161
10.2.1.	Training . . . . .	161
10.2.2.	Experiments . . . . .	161
<b>11.</b>	<b>Mobile Navigation and Manipulation</b>	<b>163</b>
11.1.	Experimental Setup . . . . .	163
11.2.	Results . . . . .	164
11.2.1.	Video: <i>composed1</i> . . . . .	164
11.2.2.	Video: <i>composed2</i> . . . . .	165
11.2.3.	Video: <i>manipulation</i> . . . . .	165
11.3.	Discussion . . . . .	165

<b>III. Conclusion</b>	<b>169</b>
<b>12. Conclusion</b>	<b>171</b>
12.1. Further Work . . . . .	171
12.1.1. Requirements . . . . .	171
12.1.2. Suggestions . . . . .	172
12.1.3. Going Further . . . . .	173
12.2. Concluding Remarks . . . . .	173
<b>Bibliography</b>	<b>176</b>
<b>Appendix</b>	<b>183</b>
<b>A. Digital Attachments</b>	<b>187</b>
<b>B. Conference Paper</b>	<b>189</b>
<b>C. Github Repository</b>	<b>197</b>
C.1. Hierarchy . . . . .	197
C.2. kmriiwa_ws . . . . .	199
C.3. kmr_behaviortree . . . . .	200
C.4. kmr_bringup . . . . .	201
C.5. kmr_communication . . . . .	202
C.6. kmr_manipulator . . . . .	203
C.7. kmr_moveit2 . . . . .	204
C.8. kmr_msgs . . . . .	205
C.9. kmr_navigation2 . . . . .	206
C.10. kmr_simulation . . . . .	207
C.11. kmr_slam . . . . .	208
C.12. kmr_sunrise . . . . .	209
<b>D. Javadoc</b>	<b>211</b>
<b>E. Operating the KMR</b>	<b>215</b>
E.1. SmartPAD . . . . .	216
E.2. Signal Units . . . . .	217
E.3. Launching an Application . . . . .	217



# List of Figures

1.1. Architecture from specialization project . . . . .	3
2.1. Sequence and selector nodes . . . . .	13
2.2. Behavior tree example . . . . .	14
2.3. The pinhole camera model . . . . .	16
2.4. Calibration board . . . . .	18
2.5. Epipolar geometry . . . . .	19
2.6. The baseline affects the field of view . . . . .	20
2.7. Feature matching . . . . .	21
2.8. Illustration of a Convolutional Neural Network . . . . .	22
2.9. Segmentation . . . . .	23
2.10. Outline of the SLAM problem . . . . .	25
2.11. Graphical representation of SLAM . . . . .	26
2.12. SLAM front-end and back-end . . . . .	27
2.13. Loop closure . . . . .	28
2.14. Lidar odometry pipeline . . . . .	30
2.15. Visual odometry pipeline . . . . .	31
2.16. SLAM block diagram . . . . .	32
3.1. KMR iiwa . . . . .	34
3.2. KMP preinstalled components . . . . .	36
3.3. Monitored areas by the laser scanners . . . . .	36
3.4. LBR iiwa 14 R820 . . . . .	39
3.5. Workspace of the LBR . . . . .	40
3.6. Interface of the media flange Touch electrical . . . . .	41
3.7. Individual motion types for the LBR . . . . .	42
3.8. Spline motion type . . . . .	43
3.9. The Robotiq 2F-85 Adaptive Gripper . . . . .	44
3.10. Gripper memory and control logic . . . . .	45
3.11. Intel Realsense Depth Camera D435 . . . . .	49
3.12. Calibration board for the Dynamic Calibrator . . . . .	52
4.1. Middleware implementation of a ROS action . . . . .	56

4.2. Architecture of Cartographer . . . . .	58
4.3. Architecture of RTAB-Map . . . . .	59
4.4. Architecture of Navigation2 . . . . .	61
4.5. Costmap cell values related to distance from robot . . . . .	63
4.6. MoveIt motion planning pipeline . . . . .	67
4.7. OpenVINO Toolkit workflow for deploying a deep learning model . . . . .	68
4.8. Creating pipelines with the OpenVINO stack . . . . .	69
4.9. Architecture of the Object Analytics pipeline . . . . .	70
5.1. Installation space for additional components . . . . .	76
5.2. Gripper adapter . . . . .	77
5.3. A D435 camera attached to the manipulator . . . . .	78
5.4. Three D435 cameras attached to the KMP . . . . .	78
5.5. Cartesian workspace for the LBR . . . . .	79
5.6. Safety configuration in Sunrise Workbench . . . . .	80
5.7. The circular box used for grasping . . . . .	81
5.8. URDF model of the D435 camera and the Robotiq gripper . . . . .	83
5.9. URDF model of KMR with devices . . . . .	83
5.10. Physical Architecture . . . . .	84
5.11. Communication Architecture . . . . .	86
5.12. Sunrise application flow diagram . . . . .	93
5.13. Implemented behavior tree . . . . .	95
5.14. Search areas for manipulator . . . . .	96
5.15. System illustration including server nodes . . . . .	98
6.1. Network response time . . . . .	107
7.1. Environment for testing . . . . .	116
7.2. Robot cells in the MANULAB . . . . .	116
7.3. Ground truth for odometry evaluation . . . . .	117
7.4. Cartographer: lidar and visual odometry separated . . . . .	118
7.5. Cartographer: lidar and visual odometry combined . . . . .	118
7.6. RTAB-Map: lidar and visual odometry . . . . .	119
7.7. Cartographer: two scans . . . . .	120
7.8. Cartographer: Three point clouds . . . . .	120
7.9. Cartographer: Two scans and three point clouds . . . . .	121
7.10. Cartographer: Five scans . . . . .	121
7.11. RTAB-Map: Two lidar scans . . . . .	123
7.12. RTAB-Map: Three RGB-D cameras . . . . .	123
7.13. RTAB-Map: Multi-session mapping . . . . .	124
7.14. RTAB-Map: 2 cameras and 2 lidars . . . . .	124
7.15. Final maps . . . . .	125



7.16. Maps of the robot cells . . . . .	126
8.1. Navigation2: Odometry velocities vs commanded velocities . . . . .	135
8.2. Navigation2: Dynamic replanning . . . . .	137
8.3. Navigation2: Dynamical adjustment of velocity . . . . .	138
8.4. Navigation2: Area for the voxel layer experiment . . . . .	139
8.5. Navigation2: Voxel layer . . . . .	139
8.6. Circular footprint of robot . . . . .	141
9.1. Experimental setup for manipulation . . . . .	150
9.2. MoveIt: Joint states of the LBR vs trajectory . . . . .	152
9.3. MoveIt: End-effector position, experiment 1 . . . . .	153
9.4. MoveIt: End-effector position, experiment 2 . . . . .	154
9.5. MoveIt: End-effector position, experiment 3 . . . . .	155
10.1. TensorBoard: total loss . . . . .	159
10.2. TensorBoard: precision . . . . .	160
10.3. Object detection: 2D detection and 3D localization . . . . .	160
11.1. Experimental setup for mobile navigation and manipulation . . . . .	164
D.1. Javadoc overview . . . . .	212
D.2. Javadoc Node abstract class . . . . .	213
E.1. Rear of a KMP . . . . .	215
E.2. SmartHMI . . . . .	216



# List of Tables

3.1.	Size of monitored fields for velocity in x-direction . . . . .	37
3.2.	Size of monitored fields for velocity in y-direction . . . . .	38
3.3.	LBR iiwa 14 R820 axis data . . . . .	39
3.4.	Relative transformation between two frames . . . . .	41
3.5.	Registers of the gripper . . . . .	45
3.6.	Inputs registers of the gripper . . . . .	46
3.7.	Output registers of the gripper . . . . .	47
3.8.	RTU Message Frame . . . . .	48
3.9.	Technical specifications of the D435 Camera . . . . .	50
3.10.	Intrinsic parameters of the D435 Camera . . . . .	51
3.11.	Extrinsic parameters of the D435 Camera . . . . .	51
5.1.	Publishers for publishing data from the KMR to ROS . . . . .	88
5.2.	Subscribers for subscribing to data from ROS to the KMR . . . . .	89
5.3.	KmpStatusdata message . . . . .	89
5.4.	LbrStatusdata message . . . . .	90



# Listings

5.1. MoveManipulator action . . . . .	88
5.2. PlanToFrame action . . . . .	101
5.3. Gripper action . . . . .	102
5.4. ObjectSearch action . . . . .	103



# Abbreviations

**AMCL** Adaptive Monte Carlo Localization.

**DWA** Dynamic Window Approach.

**EFI** Enhanced Function Interface.

**F2F** Frame-To-Frame.

**F2M** Frame-To-Map.

**FDI** Fast Data Interface.

**I/O** Input/Output.

**iiwa** Intelligent Industrial Work Assistant.

**IMU** Inertial Measurement Unit.

**IR** Intermediate Representation.

**KMP** KUKA Mobile Platform.

**KMR** KUKA Mobile Robot.

**KUKA** Keller und Knappich Augsburg.

**LBR** Lightweight Robot.

**NTP** Network Time Protocol.

**PLC** Programmable Logic Controller.

**PTP** Point-To-Point.

**RGB** Red Green Blue.

**RGB-D** Red Green Blue Depth.

**ROS** Robot Operating System.

**S2M** Scan-To-Map.

**S2S** Scan-To-Scan.

**SLAM** Simultaneous Localization And Mapping.

**SRDF** Semantic Robot Description Format.

**TCP** Transmission Control Protocol.

**UDP** User Datagram Protocol.

**URDF** Unified Robot Description Format.

**XML** Extensible Markup Language.



# Chapter 1.

## Introduction

### 1.1. Background and Motivation

Today, the world is entering the industry 4.0 revolution. Industry 4.0 encourages to data exchange, automation, decentralization and increased interaction between humans and machines. Technology changes the way the industry operates and tries to make the companies more effective, agile and profitable.

KUKA is one of the world's leading providers of intelligent robot-based automation [1]. Within Industry 4.0, mobility is an important driver. KUKA focuses on developing multiple concepts for more flexible, intelligent and mobile units that can be used within industrial production. Such a robot is the [KMR iiwa](#), hereby referred to as KMR. The KMR is a combination of a robot arm, [LBR iiwa 14 R820](#), mounted on the base of a mobile platform, [KMP 200 omniMove](#). The mobile platform and manipulator will be referred to as KMP and LBR, respectively. The platform has Mecanum wheels making it flexible and enable omnidirectional motion, and laser scanners, which makes it possible to monitor the environment. This, together with the possibilities from the installed manipulator on the base, makes the KMR meet the requirements for Industry 4.0 perfectly. Autonomous transportation is useful in applications in hospitals, for sorting of parcels in a post office or supplying parts to a work station in a production hall. Especially during this years' pandemic, Covid-19, autonomous robots could have been useful for performing tasks without the need of humans. They entail zero risk of infection and allows the production to continue running.

[ROS](#) is an open-source robot operating system that includes software libraries and tools to help developers create robot applications. It aims to be the standard that can be used by any type of robot [2]. ROS is created for distributed computing with a modular design. This, together with the flexibility given by different programming languages and operative systems, makes it a good fit for

the Industry 4.0. Integration between the KMR and ROS2, the second version of ROS, is desired. To clarify if the topic of interest is the first or second version of ROS, or ROS as a concept, the first version of ROS will be referred to as ROS1.

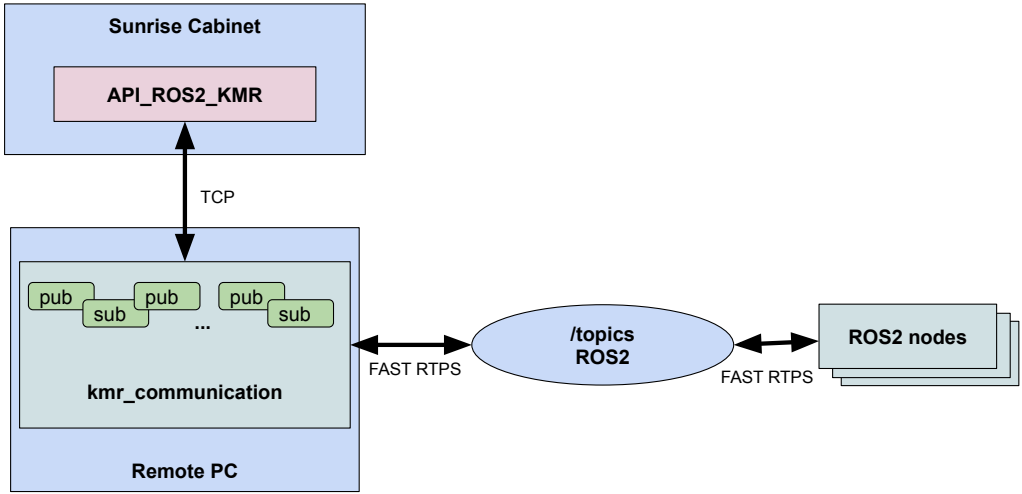
This thesis is an extension of the work conducted in the specialization project during the autumn of 2019. The objectives of the specialization project were to conduct research and get familiar with both the KMR and ROS2, and develop a simple proof-of-concept for communication between the systems. Now, this prototype should be further developed to create a comprehensive system that can be used at the Norwegian Manufacturing Research Laboratory, MANULAB, at NTNU.

Sections that describes important information about the KMR and ROS are included from the specialization thesis. This is done to deliver a comprehensive thesis presenting a complete system. Whenever sections are reused, this is clearly marked in the text.

## 1.2. Previous Work

During the fall of 2019, the authors worked on the specialization project *Configuration and Control of KMR iiwa with ROS2* [3]. The project concerned developing a prototype that integrated the operating system of a KMR with ROS2. To limit the scope, the focus was only interaction with the mobile base. The system is compatible with the ROS2 stacks Cartographer and Navigation2. The prototype has several points of improvement and potential for extensions.

The system is based on a simple architecture, shown in Figure 1.1. The architecture consists of a remote PC with ROS2 installed and the controller of the KMR, Sunrise Cabinet. The two components communicate through a TCP communication socket. The application `API_ROS2_KMR` is a Sunrise application, which is launched at the Sunrise Cabinet. It communicates with the remote PC, handling the retrieval of sensor data from the robot and executing motion commands. The application is based on a single thread running a loop where messages from the remote PC are both read and executed. The ROS node `kmr_communication` is responsible for handling the communication with the robot, processing the data received and returning motion commands. Further, the node communicates with other nodes on the ROS network from stacks such as Cartographer and Navigation2.



**Figure 1.1.:** Architecture of the prototype developed as a part of the specialization project conducted during the fall 2019 [3, p. 56].

### 1.2.1. Challenges

The focus during the project was to find the functionality for retrieving the desired data and executing the necessary commands. The performance of the ROS stacks was tested to a small extent. Several issues were discovered along the way.

The developed architecture had multiple drawbacks. A single ROS node handled the communication between the KMR and ROS2. This works for a smaller system, but is not practical as the system gets more functionality. A drawback is that if some part of the system causes an error, everything will be terminated. Another drawback is that it is not possible to launch only some of the functionality. Especially if the LBR is included in the system, it should be possible to run only the mobile vehicle or the manipulator. In addition, a large amount of code in the same files is difficult to read and handle.

Several maps obtained by utilizing the Cartographer stack were included as results. The robot cells in the MANULAB are surrounded by gratings located approximately 180 mm above the ground. The SICK laser scanners on the KMP scan the environment in 2D at a height of 150 mm above the ground. This led to unsatisfying results as the laser scanners did not detect the gratings. In addition, the Cartographer node depends on a preconfigured parameter file that was not adjusted during the project. In order to obtain satisfying maps of the environment, the parameters must be tuned and action must be taken regarding the gratings.

The same applies to the parameters of the Navigation2 stack. The parameters used are retrieved from an example project related to a Turtlebot3 robot. A Turtlebot3 robot is a small sized differential drive mobile robot, which means the parameters must be tuned to work correctly with a large and heavy robot as the KMR. During the project, a conflict between the built-in safety restrictions on the KMR and the Navigation2 controller was discovered. The SICK laser scanners monitor predefined areas around the platform. The size of the areas depends on the current velocity of the robot, where maximum velocity corresponds to the largest field configuration. If the scanners detect an object inside an area, the safety restriction of the robot is activated and stops the movement. When the KUKA Navigation Solution software controls the vehicle, the velocity is automatically adapted based on whether the monitored fields are violated. The controller of Navigation2 does not implement behavior for handling these restrictions, and this causes a conflict. When an emergency stop occurs, the navigation fails as the robot is not showing enough progress. The vehicle is not able to drive out of a violated area as all the commands from the Navigation2 controller are at a higher speed than allowed by the KMP. To make the navigation work optimally, this must be taken into account. Several solutions were suggested in the project report. They require research, but it is unlikely that the suggestions solve the problem in an acceptable manner.

### 1.3. Contributions

During the spring, the authors published a paper in the context of the 3rd International Symposium on Small-scale Intelligent Manufacturing Systems, SIMS 2020 [4]. The paper presents the system for controlling the KMR with ROS2 and verification of the system with Cartographer and Navigation2. The paper concerns the work conducted in the specialization project, together with the new architecture of the system, which was implemented during the first part of the spring. The authors will present the work at the conference on the 11th of June 2020. The paper is attached in Appendix [B](#).

### 1.4. Problem Description

The main objective of this thesis is to enable autonomous operation of the KMR in the MANULAB. This involves exploring different combinations of hardware and external software, together with implemented software.

Multiple challenges with the previously developed prototype were identified, which are included as a part of the scope of this thesis. The challenges can be split into communication, mapping and navigation.

The communication architecture should be more scalable and flexible to cope with further expansions. Fault handling must be improved to deal with both expected and unexpected events in a better way. A physical architecture for the platform must be developed to connect and assemble the external sensors and actuators considering communication and power supply. The experimental platform should be an extension of the KMR.

The environment at the MANULAB is complex with a lot of obstacles, which makes it challenging to create a good map. To improve the mapping from the specialization project, multiple cameras should be added to the mobile base. Different methods to utilize both the laser and camera data for performing [SLAM](#) should be examined.

Regarding navigation, the focus should be on how to operate the KMR safely. The built-in safety configurations of the robot need to be explored, and solutions must be found to interact with these. Compatibility with Navigation2 is previously accomplished, but the performance is not satisfactory. Further research and tuning of parameters are required to achieve autonomous navigation of the KMR.

To create a comprehensive system that exploits the possibilities of the robot, functionality to control the LBR should be added. As for the mobile base, it is desirable to utilize ROS2 for control. Including functionality for the LBR requires a thorough review of the Sunrise system to find the correct methods for moving it. The communication interface must be extended to include the manipulator.

An example of a typical area of application for a KMR is in a factory, where it can navigate around and carry objects between work stations. It is desired to accomplish such a fetch and carry scenario to combine the functionality of both the KMP and LBR. This includes the need for an overall control logic handling the task switching in the system.

Computer vision can be utilized for recognition and localization of objects. A camera is required to detect the objects and functionality for using a gripper is necessary to pick them up.

## 1.5. Related Work

Work related to operating robots manufactured by KUKA with ROS has been researched, but so far, nothing has been found regarding ROS2 or the KMR. Dömel et al. present a concept towards fully autonomous mobile manipulation using ROS1 and KUKA omniRob [5], conducted at the German Aerospace Center. The KUKA omniRob is an older version of the KMR and is built on a different control system. The project is to a small degree interacting through ROS and it

is not open source.

Sunrise applications are written in Java, which makes it possible to exploit the ROS Java client library to control KUKA robots with ROS externally. Virga and Esposito provide a ROS stack for interaction with the LBR [6], developed at the Interdisciplinary Research Laboratory at Computer Aided Medical Procedures, at the Technical University of Munich. The system is built with an architecture based on native ROS Java nodes. The nodes are launched on the robot controller, such that the two operating systems can exchange data by publishing messages utilizing the ROS framework. The stack includes functionality for a variety of different motions controls and tools for simulation as well as controlling the manipulator with the ROS MoveIt stack. The software is built on the ROS1, and according to a statement by Virga from August 2019, no plans for porting the project to ROS2 exist [7].

In December 2019, Thomas Rühr, Senior developer at Corporate Research KUKA Deutschland GmbH, presented their research on mobile manipulation with the KMR and ROS1 in the context of a ROS-Industrial Conference. The research group is working on offering a catalog of [URDF](#) models and official ROS drivers for KUKA robots, including the mobile platforms. At the moment, the group focus on ROS1, as it is still is the most widespread ROS distribution. Currently, Rühr is working on a ROSJava Adapter to control KUKA mobile platforms with ROS, similar to the system developed by Virga and Esposito.

As a part of our research, we have been in contact with Rühr via email. In the emails, Rühr states that the development is at an early stage, and points out several challenges they have faced during the implementation [8]. As a manufacturer, KUKA guarantees that their products comply to certain standards. A main problem with ROS is how to maintain single point-of-control and the strict safety restrictions of their robots. Another challenge Rühr mention is the synchronization of clocks, which is a topic in computer science that aims to coordinate otherwise independent clocks. Briefly explained, real clocks will differ after some amount of time due to clock drift, as clocks count time at slightly different rates. The challenge in this particular case is to synchronize the Sunrise clock with the ROS time. Virga and Esposito propose a solution based on [NTP](#) synchronization with a server running on the ROS master. The current pragmatic solution of Rühr is to use the ROS timestamps to overwrite the timestamps from the Sunrise system, which, according to Rühr, is a sufficient solution in their experience.

## 1.6. Outline

The overall structure of this thesis takes the form of twelve chapters, where the consecutive chapters are organized into the following parts: Fundamentals, Achievements and Evaluation, and Conclusion.

### Fundamentals

Chapters 2 to 4 concern the fundamentals of this thesis, which includes looking into the relevant theoretical background material and getting familiar with the hardware and software which are applied in this work.

Chapter 2 presents basic theory and concepts applied in the conducted work, mainly related to computer vision and SLAM.

Chapter 3 introduces the physical components of the robot system, together with their corresponding software.

Chapter 4 presents the robot operating system ROS2, together with stacks that are utilized for SLAM, navigation, manipulator motion planning and object detection.

### Achievements and Evaluation

Chapters 5 to 11 concern implementations, results and discussion of the achievements. The resulting system is comprehensive and covers a wide field of disciplines. It has, therefore, been considered a proper solution to present the components in their respective chapters.

Chapter 5 presents the developed system. This includes both the physical architecture and the implemented software.

Chapter 6 evaluates the implemented system and discusses important decisions that have been made.

Chapter 7 presents the results obtained for performing SLAM with the developed system.

Chapter 8 considers the navigation of the KMP by using the Navigation2 stack.

Chapter 9 presents planning and motion control of the LBR by using the MoveIt2 stack.

Chapter 10 evaluates the object detection approach used in the work.

Chapter 11 covers the system as a whole. The different components are composed into a comprehensive system.

**Conclusion**

Chapter [12](#) concludes the thesis, including suggestions for improvements and further work.



Part I.

# Fundamentals



## Chapter 2.

# Preliminaries

This chapter presents fundamental theory for the concepts applied in the work.

Section 2.1 presents behavior trees, which is a task switching structure commonly used in artificial intelligence and robotics. Section 2.2 introduces the field of computer vision. Computer vision is highly relevant for multiple robotic tasks, where the robot has to assess and recognize the environment. It is a broad field, and only topics important for the work conducted is included.

It is desired to perform Simultaneous Localization and Mapping, SLAM, with the implemented system. Visual Simultaneous Localization and Mapping, VSLAM, extends SLAM by applying 3D image data to describe the environment. An introduction to the SLAM problem is given in Section 2.3. Some of the content in this section is based on the theory presented in the specialization project.

### 2.1. Behavior Trees

A behavior is a way an agent thinks and acts. It can be simple, like touch an object or press play, or it can be a composition of multiple behaviors. The behaviour of walking to a location often consists of multiple sub-actions, like pass through a door, use a key if the door is locked and so on. A behavior tree is an artificial intelligence technique to structure these behaviors into plans for the agent. It is commonly referred to as a task switching structure, as it helps to decide when and to what action do we switch. Another well-known task switching structure is the finite state machines.

As the name suggests, it has a tree structure with a root node, which splits into branches with multiple nodes and ends up in leaf nodes. The nodes are executed, or ticked, in a specific order. When finished, every node returns a status: success, failure or running. The status informs the parent node about the performance

of the behavior, and gives the possibility to reason about whether to switch to a new behavior or not. If the node was succeeding, you might want to continue a sequence of actions, but if not, a fallback should be invoked. A behavior tree often implements a service-oriented architecture, where the leaf nodes of the tree work as clients and communicate with a server that performs the task.

Behavior trees have many benefits. They are user friendly to read and control, and are simple to create. The complexity will increase with scale, but a lot of graphical tools exist to make it easier to handle big trees. Dependencies between the components make it easy to change some parts of the tree without affecting the rest. The hierarchical structure makes actions exist on many levels of details, and the structure captures this in a natural way. Hence, a behavior tree is flexible, scalable and can be changed rapidly. A limitation of the more naive versions of behavior trees is that they only evaluate nodes from left to right. However, it is possible to implement other functionality like priorities and costs. A behavior tree works in a reactive manner, which means an action is just a fixed response to a given situation. This could be bad, especially in gaming, as it could be easy for an enemy to determine what your response to actions would be. This could be improved by adding good conditions and randomness to the choice of action.

### 2.1.1. Types of Nodes

A behavior tree is put together by three different types of nodes: primitives, decorators and composites.

The primitives are the most basic behaviors and consist of conditions and actions. The conditions check the state of something, like is the door open, and returns true or false based on the condition. A condition node does not change the world, and would never return the running status. Actions do something that changes the world in some way, like open the door or move agent. The primitives are also called the execution nodes, or leaf nodes, as they are always leaf nodes of the tree. They do not have any children. The primitives can be defined by using parameters, which can specify the behavior of the node. For example, a node handling walking could use input parameters to specify which coordinates the agent should walk to. While the other types of nodes are used to specify the structure and execution of the behavior tree, the primitives are providing the actual functionality of the code.

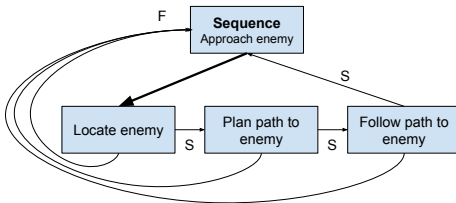
The decorators can have one child node, and their job is to change something about this. A decorator makes it possible to use action nodes for other purposes, without changing the code of the action. This adds flexibility and more control of an action. Some commonly used decorators are an inverter, which always negates the result of the child node, retry, which ticks the child again if it returns failure,

or loop, where the child node is reticked a specific number of times independently of the return.

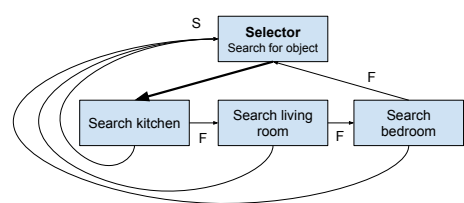
The third type is composites. The composites node describes how to group the simpler behavior types together, and it can have one to an infinite number of children. Composite nodes are typically sequences, selectors or parallels.

A sequence consists of a set of child nodes where the children need to return success for the sequence to succeed. The actions in the sequence are executed in order. If one of the nodes fails, the sequence fails immediately, and the rest of the sequence will not be executed. The child nodes are linked together by the logical operator AND. An example of a sequence could be approach enemy, which consists of locating enemy, planning a path to the enemy and follow the path to the enemy. There is no point in planning a path if you were not able to locate the enemy, and there is no point in following a path if it was not possible to plan a path. This example is illustrated in Figure 2.1a.

While the child nodes in a sequence is linked by the AND operator, a selector works as an OR gate. If one of the child nodes returns success, the selector returns success immediately, and the rest of the nodes will not be executed. This behavior is the same as an if-else statement, where the next nodes are executed only if the previous one failed. A selector could be useful in, for example, a search problem, as illustrated in Figure 2.1b. First, try searching in the kitchen, if an object is not found, search in the living room, before searching the bedroom. The selector fails if all of the children fail.



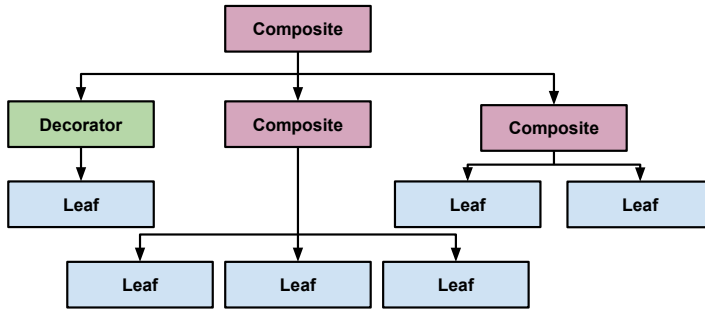
(a) Illustrative of sequence node.



(b) Illustrative of selector node.

**Figure 2.1.:** Sequence and selector nodes.

A parallel behavior node is used when multiple child nodes are to be executed in parallel. These nodes are more complex than the previous and introduce much threading. Nevertheless, it is useful in specific cases, for example, if you want to run against an enemy and shoot at the same time. It would be very impractical to have to run, stop, shoot, then start running again. For this type of nodes, a policy of when it fails and succeeds must be defined.



**Figure 2.2.:** An example of a behavior tree constituted by leaf, decorator and composite nodes.

An example of how a behavior tree could be put together by these types of nodes is shown in Figure 2.2.

## 2.2. Computer Vision

The field of computer vision concerns using mathematical models to interpret a digital image in the same way that the human brain does [9]. This includes recognizing three-dimensional structures, recognizing objects and being able to describe this in different light conditions, resolution and scales. Today, computer vision has come a long way, but it is still far from what a human can do.

Computer vision is used in a variety of fields and is important in the field of robotics. To make an intelligent robot, it has to behave and make choices like a human, which includes assessing and recognizing the environment like a human. For a robot navigation problem, computer vision is helpful for obstacle avoidance, mapping and in general exploring the environment. This could be performed with other types of sensors as well, such as laser scanners, but they tend to provide a less informative description of the environment. For a pick and place robot, advanced vision is essential to be able to identify, localize and grasp the correct objects from one place to another.

The following subsections provide an introduction to basic computer vision topics.

### 2.2.1. Projective Geometry

Euclidean geometry represents a three-dimensional space, denoted as  $X, Y, Z$ . In computer vision, the 3D world is projected into a 2D image, and it is more useful to use projective geometry. Projective geometry adds an extra dimension, denoted  $W$ . Coordinates in the projective geometry are called homogenous coordinates.

Homogenous coordinates give a way to express infinite with a finite number. A point infinitely far away in the world will always have definite coordinates in the picture. The  $W$  is basically a scaling transformation for the 3D coordinate. When increasing in dimensions,  $W$  is usually set to 1, as it then does not affect  $X$ ,  $Y$ ,  $Z$ . While Euclidean geometry gives a unique representation of a point, projective geometry is only unique up to scale. This means that only the scaling relationship between the points is known. Equation (2.1) shows how to construct a homogeneous vector from a cartesian vector, while Equation (2.2) shows how to convert a homogenous vector to cartesian.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{R}^3 \mapsto \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \in \mathbb{P}^3 \quad (2.1)$$

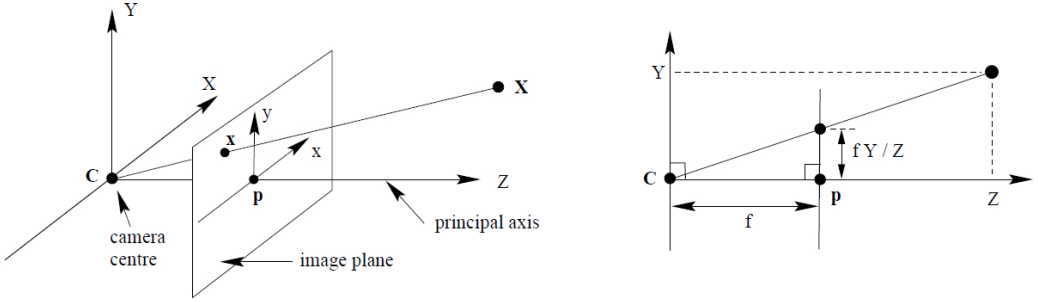
$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \in \mathbb{P}^3 \mapsto \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix} \in \mathbb{R}^3 \quad (2.2)$$

### 2.2.2. Camera Model

A camera model commonly applied in computer vision is the pinhole camera model, also called the perspective camera model. It describes the correspondence between observed points in the world and points in an image. A pinhole camera is a simple type of camera with a small aperture. The aperture is so small that from all the rays reflecting off a point in the world, exactly one enters the camera and creates a point at the image plane. This makes each point in the scene correspond to a single point in the image. Figure 2.3 shows the pinhole camera model, where a point in space,  $X$ , is mapped into a point,  $x$ , in the image plane. A straight line combines the point in space, the point in the image and the camera center. The image plane is placed in a distance,  $f$ , from the camera center. This distance is called the focal length. This geometry forms triangles, which can be used to find the projection from world coordinates to image coordinates.

The pinhole camera model is represented by the camera matrix. The camera matrix describes a matrix that can be used to map from homogeneous 3D world coordinates to 2D homogeneous image coordinates. It is shown in Equation (2.3), and is composed of two transformations, a perspective and an affine transformation. A perspective projection, handled by the extrinsic matrix, that maps the world point,  $x$ , into a normalized image plane,  $x_n$ . An affine transformation, handled by the intrinsic matrix, that maps the point in the normalized image plane,

$x_n$ , into the pixel plane,  $u$ . The normalized image plane is an image plane with a fixed position, with  $z$  set to 1, in the camera frame. Equation (2.4) shows how a point in the 3D world,  $x$ , is mapped into a pixel,  $u$ , in the image plane.



**Figure 2.3.:** The pinhole camera model.  $C$  is the camera centre and  $p$  is the principal point, or optical center. The image plane is placed in front of the camera center [10, p. 154].

$$\underbrace{P}_{\text{Camera matrix}} = \underbrace{K}_{\text{Intrinsic matrix}} \times \underbrace{\begin{bmatrix} R & t \end{bmatrix}}_{\text{Extrinsic matrix}} \quad (2.3)$$

$$u = Px = K \begin{bmatrix} R & t \end{bmatrix} x \quad (2.4)$$

The extrinsic matrix describes the pose of the world frame relative to the camera frame. Hence, it is a transformation matrix consisting of a  $3 \times 3$  rotation matrix,  $R$ , and a  $3 \times 1$  translation vector,  $t$ .

The intrinsic matrix,  $K$ , also known as the camera calibration matrix, describes the camera intrinsics. It is used to map 3D camera points to 2D pixel coordinates. Its parameters are given in Equation (2.5), and can all be found in Figure 2.3.

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.5)$$

$(x_0, y_0)$  is the principal point, also called optical center. This is where the principal axis, which is the  $z$ -axis pointing forward from the camera's projective center, intersects with the image plane.  $f_x$  and  $f_y$  are the focal length in  $x$ - and  $y$ -direction. This is the distance from the camera center to the image plane. The entry  $s$  is the skew parameter, and it defines the skew between the  $x$  and  $y$ -axis in the image. This is typically ignored, and set to 0.



A full decomposition of the camera matrix is shown in Equation (2.6).

$$\begin{aligned}
 P = & \underbrace{\begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{2D Translation}} \times \underbrace{\begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{2D Scaling}} \times \underbrace{\begin{bmatrix} 1 & s/f_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{2D Shear}} \times \underbrace{\begin{bmatrix} I & t \end{bmatrix}}_{\text{3D Translation}} \times \underbrace{\begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix}}_{\text{3D Rotation}} \quad (2.6)
 \end{aligned}$$

### 2.2.3. Camera Calibration

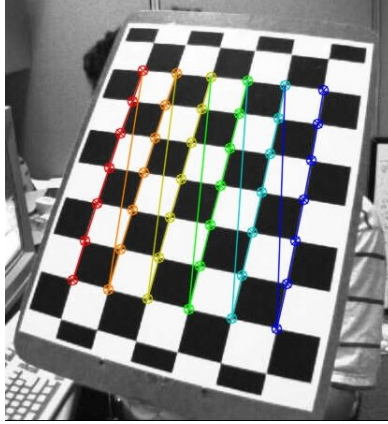
The camera calibration process concerns finding the internal and external camera quantities that affect the process of creating an image. It typically includes estimating the extrinsic, intrinsic and distortion parameters. Distortion parameters correct if straight lines in the scene do not appear straight in the image. An accurate camera model is essential when reconstructing the world model, especially for robot interaction with the world.

The general principle for camera calibration is to find the correspondence between a number of known 3D points in the world and their projection in an image. A calibration board is commonly used for this purpose. The boards are easily recognized and measured, and as the design of the board is known, errors between reality and the image can easily be detected. A calibration board often uses a chessboard pattern, as the example in Figure 2.4. In the figure, the corners of the pattern are detected by the camera. Calibration by the use of a calibration board is called targeted calibration. It is also possible to perform target-less calibration.

### 2.2.4. Stereo Vision

Stereo vision is the process of using two cameras to find a 3D representation of an image. This is done by finding correspondent pixels in the two images and projecting their positions into 3D [9, p. 469]. This is the same way the human perception work, where the depth is found based on the differences between the left and the right eye. Without a second camera with a known relative position, it is not possible to determine the 3D position of the points in the image plane as the depth is unknown.

The problem of finding correspondent pixels in two images is called the correspondence problem. Geometric relations between the 3D points and their projection onto the 2D image planes of each camera make up constraints. These relations



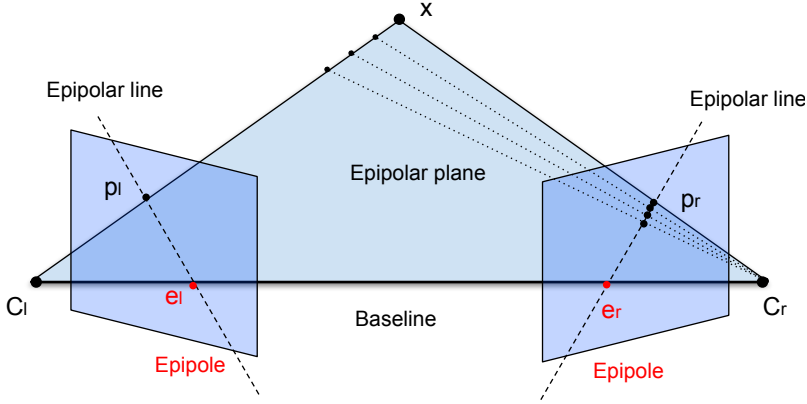
**Figure 2.4.:** A typical calibration board glued to a rigid surface. The colored markers denote detected corners [11].

constitute what is called the epipolar geometry, and the information is implicit in the relative pose and calibrations of the cameras [9, p. 472]. In favor of a regular pixel matching problem, stereo matching has the advantage of knowing both the relative position between the cameras and the calibration parameters of the cameras. The geometric constraints make the search for alternative pixel matches largely reduced. The relations are deduced by using the assumption that the cameras can be approximated by the pinhole camera model.

The following list describes terms related to epipolar geometry. These entities are visualized in Figure 2.5.

- The **baseline** is the line connecting the two camera centers.
- The **epipolar plane** is the plane bounded by the baseline, hence the two camera centers, and the point of interest.
- An **epipole** is the intersection point between the image plane and the baseline.
- An **epipolar line** is the intersection line between the epipolar plane and the image plane.

A point in the world,  $x$ , is projected into the left image plane to a point  $p_l$ . This point could have come from any point on the projection line between  $x$  and  $p_l$ . Each of these points projects to possible  $p_r$  points in the right image plane. This is illustrated in Figure 2.5. The possible points lie along the epipolar line in the right image. This means, for each image point in the left image plane, the corresponding point in the right image plane must be on the known epipolar line. This constraint is called the epipolar constraint. Epipolar constraints can also be described by the



**Figure 2.5.:** The epipolar geometry.  $C_l$  and  $C_r$  denotes the left and right camera center.  $x$  is the point of interest,  $p_l$  and  $p_r$  are the point of interest in the image plane and  $e_l$  and  $e_r$  denote the left and right epipole.

essential or the fundamental matrix. Both matrices are  $3 \times 3$  matrices describing point correspondence between two points in two images. The essential matrix,  $E$ , describes the correspondence between points in normalized image coordinates, while the fundamental matrix,  $F$ , relates points in pixel coordinates. To achieve normalized image coordinates, the intrinsic parameters of the camera must be known. The origin of the coordinates is at the optical center, and the focal length is used to normalize the  $x$  and  $y$  coordinates.

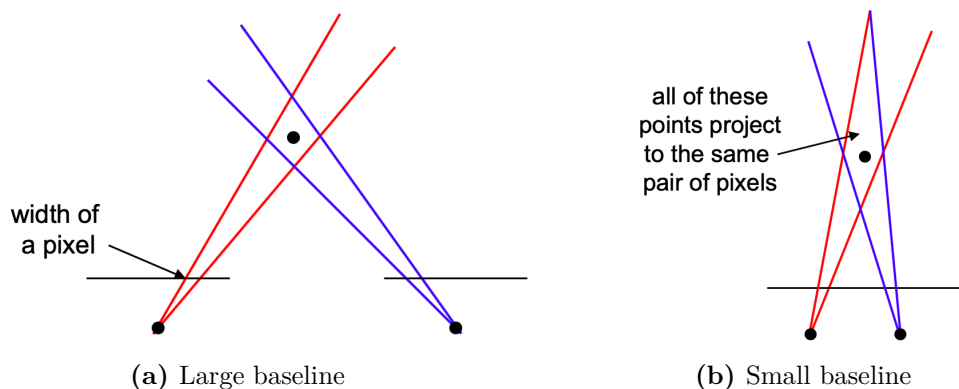
The two matrices are related as follows:

$$E = (K')^T F K \quad (2.7)$$

where  $K'$  and  $K$  is the intrinsic calibration matrix of the cameras involved.

As the essential matrix includes the intrinsic parameters of the camera, it has only five degrees of freedom. Thus, it is easier to find than the fundamental matrix with seven degrees of freedom.

The baseline between the cameras is highly affecting the total field of view, FOV. Therefore, it is important to find the right baseline when designing a camera with stereo vision. A short baseline gives a bigger FOV leading to more uncertainty regarding the depth. A wider baseline gives a smaller FOV, which gives a better distance estimation. This is illustrated in Figure 2.6. On the other hand, it is harder to find matching pixels and the search problem gets more complex, leading to lower quality of the result.



**Figure 2.6.:** The size of the baseline affects the field of view. A large baseline gives a smaller field of view, while a short baseline gives a larger [12].

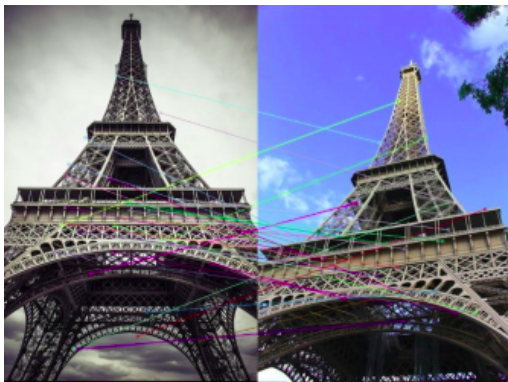
### 2.2.5. Image Rectification

Image rectification is an important part of the correspondence problem. The epipolar constraint causes the search for corresponding pixels to be limited to only the epipolar line instead of the whole picture. If the images are rectified, which means aligned to be coplanar, the search is limited to one dimension. This dimension is a horizontal line parallel to the baseline. A one dimensional search is faster and reduces the possibility of wrong matches.

### 2.2.6. Features

In computer vision, features are informative parts of images relevant to computer vision applications. Examples of features are edges, corners, blobs, ridges or whole objects. Feature detection is a low-level image processing operation which localizes features in images. A well known feature detection algorithm is the Harris Corner detector [13]. The idea is to pass a sliding window over each pixel of an image, searching for differences in intensity for a small displacement in all directions. Pixels that exceed a certain threshold and are local maxima within a particular window are classified as corners. Feature extraction is the process of computing descriptors for the detected features and is related to dimension reduction of images. A feature descriptor contains encoded information about a feature, and can be used to differentiate one feature from another. An important characteristic of a descriptor is that it is invariant to image transformations, such that a feature can be recognized in images taken from different points of view. Feature matching can be performed based on these descriptors, which is useful in applications like object classification, or for computing relative transformation between two camera frames. An example of feature matching is shown in Figure 2.7. The motive is

the same, but the adjustments and viewpoints of the images differ.



**Figure 2.7.:** Feature matching in images of the same motive considered in different conditions [14].

### 2.2.7. Segmentation

An image can be partitioned into segments, each containing important information for further processing of the image. Segmentation creates a mask for each object in an image, where each segment is a group of pixels with similar attributes. Region-based segmentation differs objects from the background by searching for a sharp contrast in pixels. This can be done by calculating the average pixel value for a greyscale image and apply it as a threshold, known as threshold segmentation. Objects with pixel values higher than the threshold are denoted as foreground, while lower valued pixels are classified as background. Edge segmentation finds the boundaries of objects in an image by searching for regions separated by edges by feature detection. Clustering techniques known from machine learning can also be applied for segmentation. The process divides pixels of an image into several clusters, where the points in each cluster are more similar to other points in that same cluster than those in other clusters.

### 2.2.8. Object Detection

Object detection is one of the most complex challenges of computer vision and concerns recognition and localization of real-world object instances in images. The general term object detection can be broken down into a collection of computer vision related tasks that involve identifying objects in images.

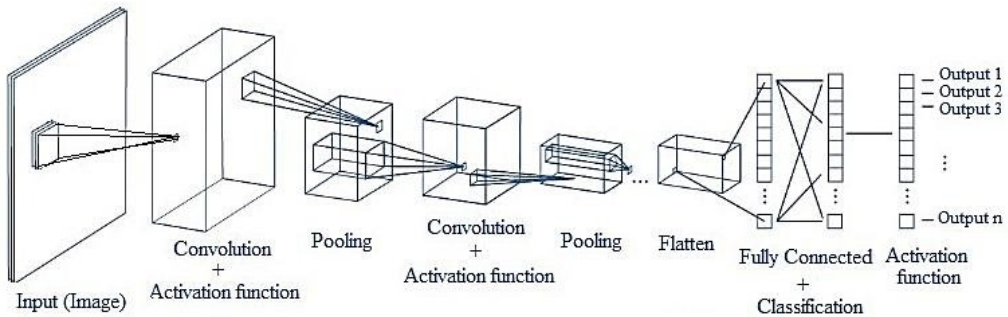
Image classification involves predicting the class of an object in an image based on predefined class labels.

Object localization refers to identifying the location of the detected objects in an image and drawing a bounding box around it. A bounding box is an imaginary box that embraces the detected object and can be defined in 2D or 3D, indicating the pose of the instance.

Object detection combines the two tasks, and locate the presence of objects with a bounding box along with the class for each of the located objects in the image. A relevant approach for each of the problems is described in this section.

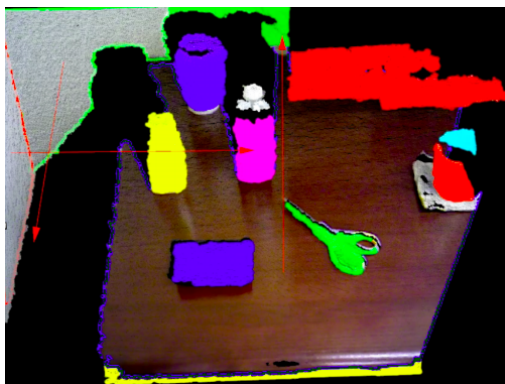
## Image Classification

A Convolutional Neural Network, CNN, is a deep learning algorithm commonly applied in computer vision. The network takes an image as input, assign importance to objects in the image and is able to differentiate objects from others. The output can be a single class or a probability of classes that best describes the image. A computer can perform image classification by searching for features such as edges and curves, and then building up to more abstract concepts through a series of layers. A CNN consists of an input and an output layer, as well as multiple hidden layers.



**Figure 2.8.:** Illustration of a Convolutional Neural Network [15].

Figure 2.8 shows an example of a typical CNN. The first layer in a network is always a convolutional layer, which is designed for detection. The convolution layer passes a filter, or a kernel, over the image and extract features. The filter is an array of numbers, known as weights. The pooling layers downsample the extracted information to retain only the most important information. Commonly, the network layers alternate between convolution and other types of layers, such as pooling layers. The hidden layers include nonlinear activation functions that enable the network to learn and model nonlinearity.



**Figure 2.9.:** An example of a segmented desk scene. Planar surfaces are outlined in color. The red arrows indicates the normal direction of the segmented planes [16].

The output layer is called a fully connected layer, which based on the features extracted throughout the network, determines to which class the features mostly correlate.

Initially, the weights or filter values of the network are randomized, and the layers are not able to extract features. An evaluation method for how well the network models the given data is by a loss function. It is desired to minimize the loss for a neural network by optimizing its weights. The process of adjusting the layers for the defined classes is through a process referred to as training of the network. For the training process, a data set consisting of images with corresponding class labels are applied. Similarly, a test data set of images and labels can be constructed to validate that the network works as desired when the training process is finished.

## Object Localization

Object localization attempts to identify the location of an object in an image with known class label and region of interest in 2D. Given a point cloud, segmentation can be applied to detect the boundaries of the object within a 3D region. One such algorithm is the Organized Multi Plane Segmentation [16], which attempts to segment points present in a point cloud.

An example of a segmentation process of a table with objects is shown in Figure 2.9. The algorithm starts by calculating normals for each point. Further, planes are estimated from planar regions in the point cloud with similar normals. Then, clusters that connect points that are similar based on criteria such as color, depth and normals are created. This results in segments of objects and planes. In order to localize the 3D objects in space, contiguous clusters of points above

planar surfaces, such as the table, are extracted. By only considering the points of interest, according to the 2D region of interest, the corresponding segmentation in three dimensions of the object can be retrieved. A bounding box surrounding the segmented points describes the pose of the object in three dimensions.

## 2.3. Simultaneous Localization and Mapping

SLAM, Simultaneous Localization And Mapping, is the process of mapping an unknown environment while localizing relative to the map. The goal is to use the environment to update the pose of the robot. SLAM exists in many varieties, and a selection of approaches are presented in this section. As the KMR only moves in the plane, only 2D SLAM is considered.

The requirements of the problem is a mobile robot, a device to provide measurements of the environment and transformations defining the position of the sensors in relation to the base of the robot. This section will present both lidar and visual SLAM. Lidar SLAM uses lidars, either 2D or 3D, to map an unknown environment. Visual SLAM, denoted VSLAM, concerns creating a map by the use of camera sensors and computer vision.

Odometry is the use of data from sensors to estimate the change in position over time, where the sensors can be wheel encoders, [IMUs](#), lidars or cameras [\[17\]](#). As the odometry from the wheel encoders often is erroneous, the range measurements can be used to correct the position of the robot. The motion of the robot, relative to the world, can be reconstructed by tracking how the environment appears to move through sequences of measurements. This process is described in [Section 2.3.6](#) for both 2D and 3D input data.

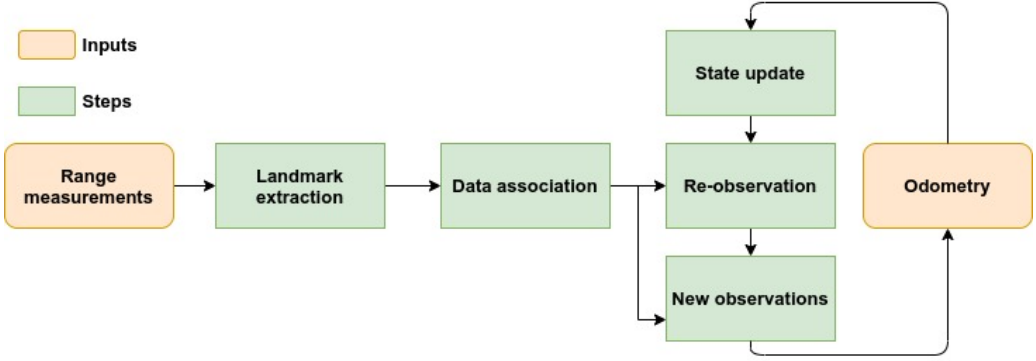
In general, SLAM consists of the steps landmark extraction, data association, state update and processing of new observations, as illustrated in [Figure 2.10](#).

The pose of the robot is updated by measuring the environment and observing how the robot moves. When the robot moves, the uncertainty in the pose is updated. Features are extracted from the range measurements of the environment. The data is associated and matched with previous observations. Re-observed landmarks are used to update the robot's pose and new landmarks are added as new observations in a submap of the current state.

### 2.3.1. Map

A map is an organized representation of submaps containing observed landmarks in the environment. A submap is a local representation of a scene with a predefined size. Landmarks are stationary features that can easily be re-observed and





**Figure 2.10.:** Outline of the SLAM problem with input and steps of the process.

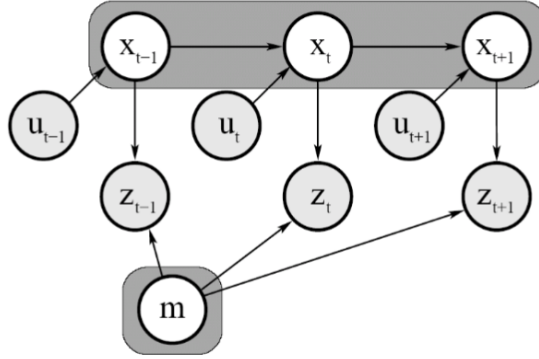
distinguished from the environment. When calculating a map of the environment, it is assumed that the robot's odometry and observations are known, while the pose of the robot and the poses of landmarks are unknown. A typical map format for navigation applications is the occupancy grid map. In an occupancy grid map, the environment is represented as discrete grids, cells, where each grid has a value based on whether it is occupied or not. Assuming the pose of the robot is known, a posterior probability for each cell is computed based on range measurements from sensors. The value of the grid represents the degree of occupancy where 100 is fully occupied, and 0 is free. If the occupancy of a cell is unknown, it has the value -1.

### 2.3.2. Online and Full SLAM

The SLAM problem can be classified as online SLAM or full SLAM. Online SLAM attempts to recover only the most recent pose of the robot. Full SLAM estimates the entire path of robot poses after all data has been gathered, and can be described by the probabilistic term:

$$p(x_{0:T}, m | z_{1:T}, u_{1:T}) \quad (2.8)$$

Equation (2.8) represents the probability of a path,  $x_{0:T}$ , and a map,  $m$ , given the range measurements,  $z_{1:T}$ , and the odometry,  $u_{1:T}$ , of the robot. The uncertainty in the state, or robot pose, and the uncertainty of the observed landmarks are kept track of and updated as new measurements arrive. The full SLAM problem can be conveniently modelled and expressed with a graph representation as in Figure 2.11.



**Figure 2.11.:** Graphical representation of the SLAM problem [18].

A high probability is desired for the estimates of the map and the states of the robot, which makes Equation (2.8) an optimization problem.

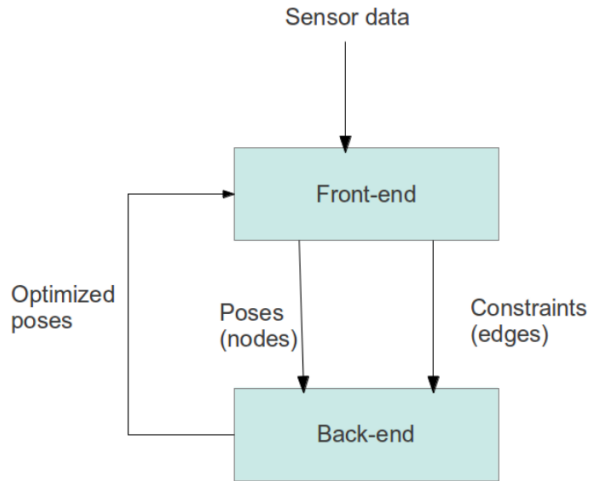
### 2.3.3. Filter and Optimization SLAM

Approaches to solve SLAM can be based on either filtering or optimization methods. In a filter-based approach, the information about the environment and the states of the robot are maintained as a probability density function. The estimation and update steps are performed recursively. Kalman filters and particle filters are known approaches to this variant of SLAM. The optimization-based approaches use a graph structure to represent the robot measurements and poses and are commonly referred to as graph-based SLAM. The nodes represent states and measurements acquired at the poses. The edges represent constraints consisting of the relative transformation between two poses. The optimization process concerns finding the configuration of robot poses that best satisfies the constraints. The transformations are either odometry measurements between two robot positions or determined by aligning the observations acquired at the two locations. The latter is known as bundle adjustment. It is defined as the process of refining the 3D coordinates describing the scene and the transformations between a set of images taken from different viewpoints. The graph-based approach is a commonly applied method in visual SLAM [19], and is the one emphasized in this work.

### 2.3.4. Local and Global SLAM

The SLAM process can be decomposed as front-end and back-end subsystems, as shown in Figure 2.12. Local SLAM is the front-end component that builds a succession of submaps. The back-end subsystem performs global SLAM in

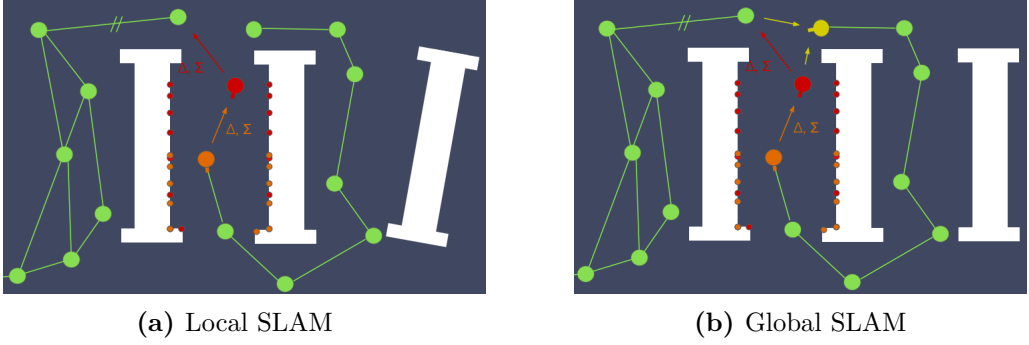
parallel, which means detecting loop closure constraints and optimizing the pose graph.



**Figure 2.12.:** SLAM systems decomposed by the front-end and back-end components [19, p. 124].

The local submaps created in the front-end include data associations that only depend on recent scans. When a specified amount of data is inserted into the submap, a new submap is created. The submaps are locally correct, but they might drift over time, as illustrated in Figure 2.13. When the red robot moves to the green position, it creates a constraint defined by the relative pose  $\Delta$  and the covariance matrix  $\Sigma$ . Range measurements are illustrated as red dots on the white racks. As each submap only contains a few observations, the localization error will grow over time. In Figure 2.13a, drift has occurred, and the righter most rack seems tilted in the map. This is fixed by global SLAM, also known as loop closure. When the robot comes out of the area between the racks, the robot attempts to recognize features based on every feature previously observed. The goal is to align the submaps and create a global map. Loop closure makes it possible to retain the correct topology of the environment and to reduce the uncertainties in the pose and landmark estimates.

The global map is created as a topological graph where the submaps are nodes and the edges are their relative alignment to each other. This is a nonlinear optimization problem where the edges are to be optimized to reduce the error within the cycles that occur in the graph. The optimization problem can be run in batches when there are enough available submaps in the graph.



**Figure 2.13.:** The small red dots corresponds to range measurements, while the larger circles corresponds to positions of the robot. In Figure 2.13b a loop closure is detected between the red and the yellow circles. Loop closure correct the true topology and reduce errors due to drift that occur over time [20].

### 2.3.5. Maximum A Posteriori Estimate

Global SLAM concerns estimation and inference on the data received from front-end. The problem is formulated as the estimation of a set of unknown state variables,  $X$ , that includes both the pose of the robot and the structure of the environment.  $Z$  is a set of noisy sensor measurements that depend on the true state  $X$ . The Maximum A Posteriori estimate, MAP, defined in Equation (2.9), is a commonly used estimator for  $X$  [21].

$$\begin{aligned}
 X^{MAP} &= \operatorname{argmax}_X p(X|Z) \\
 &= \operatorname{argmax}_X \frac{p(Z|X)p(X)}{p(Z)} \\
 &= \operatorname{argmax}_X l(X; Z)p(X)
 \end{aligned} \tag{2.9}$$

Equation (2.9) maximizes the posterior probability,  $p(X/Z)$ , of the states  $X$  given the measurements  $Z$ . After applying Bayes's law, the resulting equation is a function of  $l(X; Z) \propto p(Z/X)$ , which is the likelihood of the states  $X$  given the measurements  $Z$ . The likelihood can be seen as a sensor model that evaluates how well the measurements fit the current estimate for the states  $X$ . The density  $p(X)$  is the prior density over the states, which evaluates how well the current estimate fits with any prior knowledge about  $X$ . Nonlinear least square methods can be applied to find the MAP estimate for the states  $X$ , given noisy measurements and corresponding measurement prediction models. The full derivation is given in [21], but summarized is the goal of the process to minimize a measurement error function, which is the difference between the measurement prediction function

and the measurement.

The measurement error function in the MAP estimate is based on either a geometric or a photometric error. The geometric error measures the difference between the measured pixel correspondence and the predicted pixel position, given the current estimation of the robot pose and an observed point in the 3D scene. The photometric error directly compares intensities in two frames to determine the transformation. In VSLAM, the pose estimation can be classified as either indirect or direct. Indirect approaches, also known as feature-based methods, start by finding feature correspondences and use these to optimize the geometric error. Direct approaches skip the feature extraction step and use image intensity to optimize the photometric error directly.

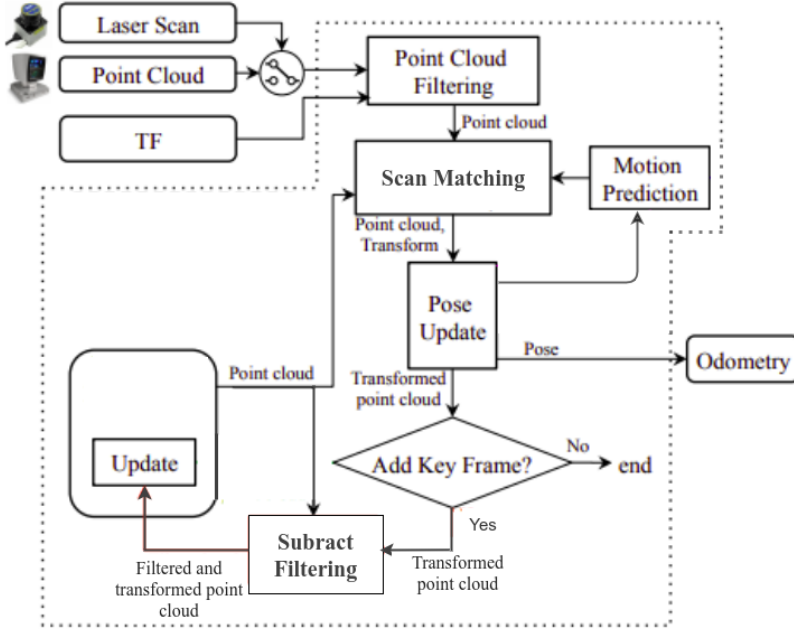
### 2.3.6. Localization

Localization relative to a map is performed by using odometry and transformations between measurements. Odometry can be calculated from any sensors capable of estimating the change in position over time. Lidar odometry estimates the change in position over time by comparing a succession of scans. Similarly, visual odometry uses cameras to estimate the motion by comparing frames. The problem is modeled as the minimization of a nonlinear least squares problem. The parameter to determine is the transformation matrix, corresponding to the motion between two measurements. Two different approaches are distinguished by whether to match new data against the previous data received or a local map. For lidar odometry, new scans are compared either to the last scan, Scan-To-Scan (S2S), or to the local submap created from previous scans, Scan-To-Map (S2M). The same concept applies for visual odometry, where new frames are compared either to the previous frame, Frame-To-Frame (F2F), or submap, Frame-To-Map (F2M), considering 3D features instead of scans. The Frame- or Scan-To-Map approach will be further emphasized.

### Lidar Odometry

Lidar odometry is the process of finding a transformation between two scans measuring the same scene in order to estimate the motion of the robot. The process is depicted in Figure 2.14 and described as following. The input point clouds are filtered, or downsampled, and normals are computed. Normals are necessary to be able to recognize planes in the space. Points are aligned in the point cloud map by considering nearby points or planes. The correspondences are unknown, and the matching requires a motion prediction to estimate a transformation. This prediction can be based on a motion model based on previous point registrations

or odometry from other sensors. The pose is updated from the calculated transformation, and the odometry is corrected with a corresponding covariance. If enough points correspond in the matching module, the points are added to the point cloud map. This is done by subtracting the map from the new point cloud, only leaving new points which are further added to the point cloud map.

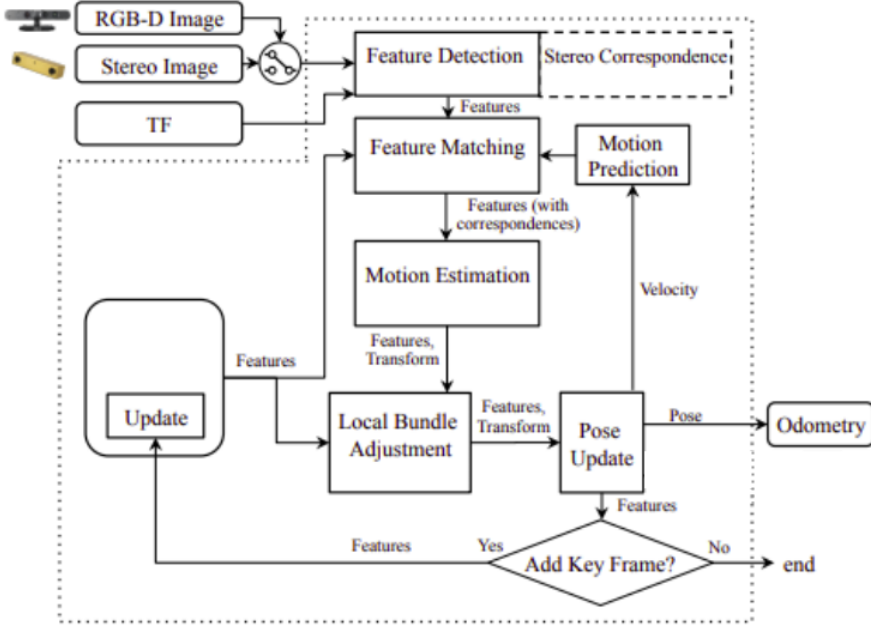


**Figure 2.14.:** Block diagram showing the lidar odometry pipeline. TF (transformation) defines the position of the camera in relation to the base of the robot [22].

## Visual Odometry

Visual odometry is the process of finding a transformation between two image frames, measuring the same scene, in order to estimate the motion of the robot. The process is depicted in Figure 2.15 and described as following. Features are detected in the input image data, which are further matched with features in the local map. A motion model can be used to estimate where in the current frame the already contained features should appear. In this way, the search window for corresponding features is limited. The motion is estimated by computing the transformation of the current frames accordingly to the features in the map. For a transformation to be accepted, a minimum number of matches are required. Further, the transformation is refined by local bundle adjustment. The odometry

is updated based on the estimated transformation, with a corresponding covariance. The extracted features from the new frame with no matches in the map are added to the local submap. For matched features, the position of the features in the map is refined based on the transformation obtained from bundle adjustment.

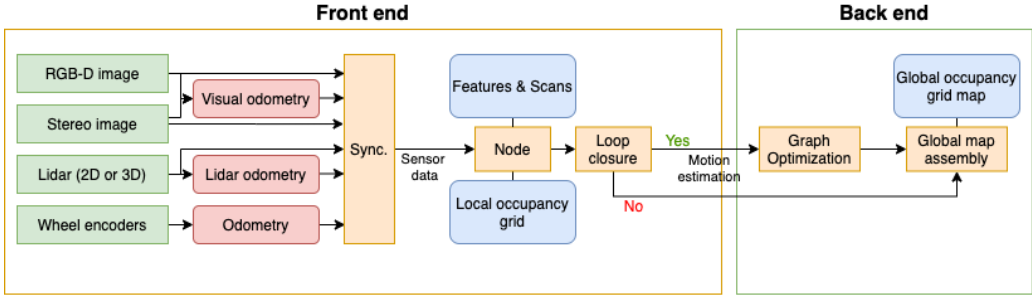


**Figure 2.15.:** Block diagram showing the visual odometry pipeline. TF (transformation) defines the position of the camera in relation to the base of the robot [22].

### 2.3.7. Mapping

The following description of the mapping process is based on the creation of a 2D occupancy grid map using a graph-based approach. Graph-based SLAM is built on a structure of nodes with edges representing relative poses and confidences. A block diagram of a graph-based SLAM system is shown in Figure 2.16. Each node stores observations and a local submap describing the environment for its estimated pose. The graph in total represents a global map of the environment.

A local occupancy grid is computed from a few consecutive sensor measurements, where the grid cells are either denoted as empty or occupied cells. New sensor data is continuously processed and aligned to the local grid map, where the processing of the input depends on the type of data. For 2D lidar data, each ray is processed, and if hitting an object the cell is denoted as occupied. The cells



**Figure 2.16.:** Block diagram of graph-based SLAM system.

between the lidar and the obstacle are denoted as empty. It is assumed that the ray is parallel to the ground, such that it can not detect the floor. 3D depth images are projected in 3D space and transformed, resulting in a point cloud in the robot base frame. A segmentation algorithm eliminates points corresponding to the ground. The normals of the point cloud are computed, and points with normals pointing upwards within a certain angle are labeled as horizontal planes. Further, segmentation is performed on planes close to the ground, according to the robot coordinate system, to recognize and eliminate the plane corresponding to the floor. The remaining points are denoted as obstacles. The points are further projected and merged into 2D space. Cells with points are denoted as occupied and cells between the sensor and the obstacle, including ground points, are denoted as empty. New measurement data are initially inserted into the local submap according to a pose estimate based on the odometry. The alignment can further be refined through a scan matching process to optimize the local occupancy grid.

When a local occupancy grid no longer changes, it is considered as finished, and a new submap is started from new measurements. A new node is established in the graph with the local occupancy grid, measurements of the environment and its estimated relative pose. Features and scans from the sensor measurements, which are to be used for matching, are kept in the node. The local occupancy grid of the node is transformed into the global occupancy grid based on its pose estimation. The node's observations are compared to nearby, previously created nodes in the graph. Edges are added to the graph representing a transformation between two nodes, either as subsequent nodes or through loop closure detection. The transformation is computed based on the motion estimation approach described for visual odometry. If a loop closure is detected, a graph optimization approach is applied to minimize the residual errors in the graph introduced from the cycle. The computed error is propagated through the whole graph, re-assembling the global map according to the optimized poses for all of the nodes.



# Chapter 3.

## Hardware

This chapter comprises the physical components that this work concerns, together with their corresponding software.

A comprehensive description of the KMR was given in Chapter 3 of the specialization project [3]. A selection of the content from previous work is reproduced and complemented in this chapter. Section 3.1 describes the KMR in general. Section 3.1.3 presents the mobile platform, KMP, including components and how the vehicle can be configured. Section 3.1.4 presents the manipulator, LBR, including how the robot arm can be programmed for motion. Motion programming of the platform is omitted as it is not a part of the scope of this thesis.

The work includes extending the KMR robot system with a gripper and cameras. A Robotiq 2F-85 gripper is described in Section 3.2. Section 3.3 presents the Intel Realsense D435 depth camera.

### 3.1. KMR iiwa

KMR is a designation for mobile robot systems from KUKA, and iiwa is short for intelligent industrial work assistant. The KMR iiwa, shown in Figure 3.1, is composed of the KMP 200 omniMove and the LBR iiwa 14 R820 [23].

The intended use of the mobile robot is to handle automated manufacturing tasks and transport components. It can be moved around freely, link solitary production cells and conduct industrial tasks, resulting in versatile and effective production units. The manipulator can only be moved when the platform is stationary, and the platform can only be moved when the manipulator is stagnant in a specific position. This position is located over the base, hereafter referred to as the drive position.

Sunrise Cabinet is the control hardware of the KMR and is contained inside the



**Figure 3.1.:** KMR iiwa is composed of a mobile platform and a lightweight manipulator [24].

mobile base. There are two PCs inside the Cabinet, the KUKA control PC and the Navigation PC. Motions executed by the KMR is controlled by the Cabinet, and it monitors the kinematic systems of both the KMP and the LBR.

The KUKA smartPAD is a teach pendant with a touch-sensitive display that can be used to jog the KMR manually and to launch applications. It must be connected to the platform by cable.

### 3.1.1. Operating Modes

There are three main operating modes for the KMR, T1, T2 and AUT. The manual operating modes, T1 and T2, can be used for testing of programs. T1 is the manual reduced velocity mode, and T2 is the manual high velocity mode. In manual mode, the smartPAD can be used to operate the KMR by either jogging the robot or by executing an application. When jogging the robot, it is moved according to the robot coordinate system. In autonomous, AUT, mode, the platform can be operated by KUKA NavigationSolution software or by an application. Before the KMR can be controlled in AUT mode, the *PositionAndGMSReferencing* program needs to be launched to calibrate the sensors of the LBR. The position referencing checks whether the saved zero position of the motor of an axis corresponds to the actual mechanical zero position [25, p. 319]. A description of how the KMR is operated can be found in Appendix E.

### 3.1.2. Software

KUKA Sunrise OS is a system software for industrial robots, operated by the Sunrise Cabinet. Sunrise OS offers functionality for programming, motion control and configuration of advanced robot applications. The software is installed on an external Windows 7 PC, referred to as the work computer, which connects to the KMR via ethernet.

Programming related to the Sunrise components of the system is performed in Sunrise Workbench on the work computer. Sunrise Workbench is a Java-based development environment for developing robot applications and managing Sunrise projects. A Sunrise project contains the necessary data for the operation of a robot system and is configured and connected to a specific robot. The safety configuration file of the project contains safety functions preconfigured by KUKA and can be edited in Sunrise Workbench. The I/O configuration file of the project contains the complete bus structure with associated I/O mappings of the KMR, which can be edited in KUKA WorkVisual. WorkVisual is a program for I/O configuration and mapping coordinated with the robot controller and the PLC for motion and safety control [25, p. 217].

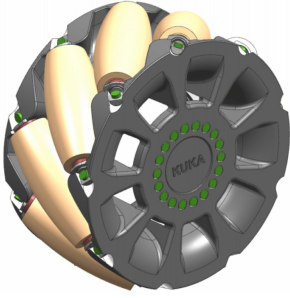
A Sunrise project can contain Sunrise applications as well as regular Java files. Sunrise applications define the tasks that are to be executed at the controller, and only one application can be executed at the controller at a time. Sunrise background applications can be executed on the controller parallel to a Sunrise application, to perform monitoring of sensors or control tasks for external devices.

Sunrise applications are based on KUKA RoboticsAPI and are characterized by the extension of a RoboticsAPIApplication. KUKA RoboticsAPI is a Java interface for controlling KUKA robots and external devices. It contains predefined data types that are required for programming KUKA robots and are based on the object-oriented Robotics API [26], for developing software for industrial robotic applications. KUKA RoboticsAPI allows general specifications of motions, devices, control, handling requests and I/O operations. Additional software for controlling and programming the KMP includes Sunrise Mobility and KUKA Navigation Solution. Mobility extends KUKA RoboticsAPI with platform-specific software for configuring and programming mobile platforms. Navigation Solution is an optional software package that provides functionality for autonomous navigation and SLAM for mobile platforms.

### 3.1.3. KMP 200 omniMove

The mobile platform KMP works as a transporter of the LBR and for any products to be carried. The KMP is equipped with four Mecanum wheels, and two

SICK S300 Expert laser scanners, shown in Figure 3.2. The mobile platform can be moved omnidirectionally, in both positive and negative x- and y-direction and rotated around the z-axis. It can also be moved in multiple directions simultaneously, that is, in diagonal direction or moving around curves by superimposing linear motion and rotation.



(a) Mecanum wheels [23, p. 31]

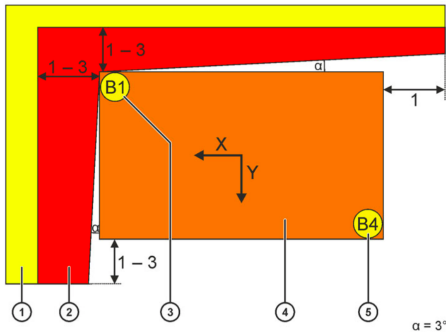


(b) SICK S300 Expert laser scanner [27]

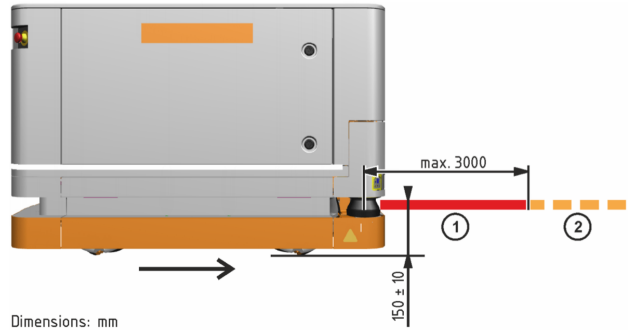
**Figure 3.2.:** Preinstalled components on the KMP 200 omniMove.

### SICK Laser Scanners

The SICK S300 Expert is a compact laser measurement system that scans the environment in two dimensions by using infrared laser beams. The scanners can measure the distance to obstacles up to 30 meters away with the time-of-flight principle [28, p. 123].



(a) Structure of the monitoring range [29, p. 115]



(b) KMP with the monitoring range [23, p. 44].

**Figure 3.3.:** KMP with laser scanners and the monitored areas.

As shown in Figure 3.3a, the lasers B1 and B4 are positioned diagonally opposite

one another. Each scanner has a range of  $270^\circ$ , covering one long side and one short side of the vehicle. The front side (+X) and the right-hand side (-Y) are covered by scanner B1, while B4 covers the rear side (-X) and the left-hand side (+Y). The lasers scan at the height of  $150 \pm 10$  mm above the ground, as illustrated in Figure 3.3b.

The laser scanners have two functions. The primary purpose is to operate as the safety equipment of the system by monitoring predefined areas around the vehicle. The second function is to provide laser range data for autonomous navigation with KUKA Navigation Solution. The monitored areas are divided into two fields, the warning field and the protective field. The size of the two fields depends on the velocity and the direction of the platform. For example, if the vehicle drives forwards, only the fieldsets of laser scanner B1 are active. The maximum size of the protective field is 3 m. Table 3.1 and 3.2 shows the size of the monitored fields for different velocities in X and Y direction, respectively<sup>1</sup>. When the KMR is operated in T1 or T2 mode, and the velocity of the platform is less than  $0.13 \frac{m}{s}$ , the laser scanners are inactive [23, p. 70]. Inactive in this context means that the protective field and warning field are not monitored. The laser scanners are always active in AUT mode and provide scans that are processed by the Navigation PC. The behavior of the system, when these fields are violated, depends on the selected operation mode. If the warning field is violated in AUT mode, the maximum velocity of the vehicle is reduced to 100 mm/s. The vehicle stops if the protective area is violated independently of the operation mode.

**Table 3.1.:** Size of monitored fields for velocity in x-direction [30, p. 121]

Velocity [mm/s]	Protective field [m]	Warning field [m]
100	0.45	0.49
200	0.55	0.63
300	0.66	0.78
400	0.76	0.92
500	0.87	1.21
600	0.97	1.36
700	1.08	1.36
800	1.18	1.50
900	1.29	1.65
1000	1.21	1.61
1100	1.32	1.76

<sup>1</sup>The values are taken directly from the documentation of Sunrise Mobility [30]. The values for the field sizes for a velocity of 1000 mm/s in X-direction seems to be incorrect.

**Table 3.2.:** Size of monitored fields for velocity in y-direction [30, p. 121]

Velocity [mm/s]	Protective field [m]	Warning field [m]
100	0.43	0.47
200	0.54	0.62
280	0.62	0.73
300	0.64	0.76
400	0.75	0.91
500	0.87	1.07
620	0.91	1.16

### Safety Bus System

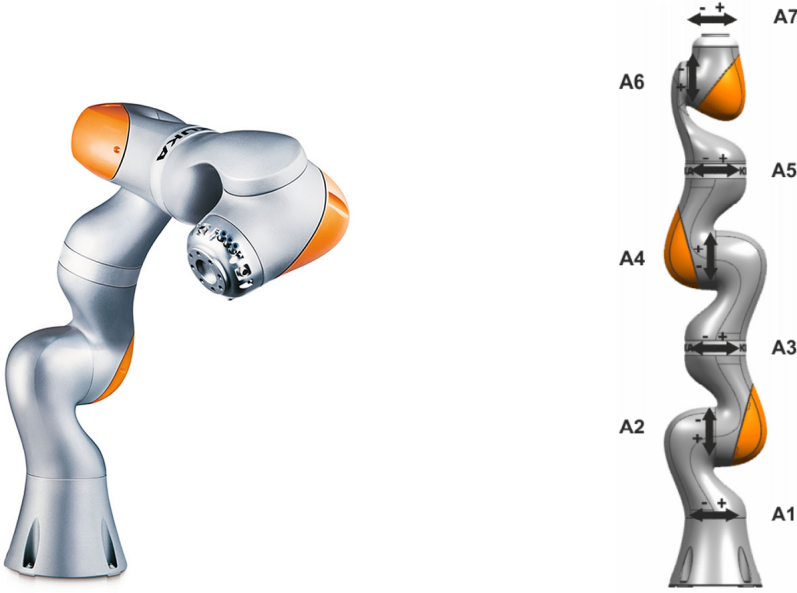
The safety functionality of the mobile platform is controlled by a Siemens PLC. The laser scanners are integrated with the aid of an EFI gateway to enable the bidirectional transmission of data between the lasers and the PLC. The PLC receives data from the lasers, which are processed and further sent to the controller. If the protective or warning field is violated, safety events are triggered on the controller, such as stopping the vehicle or reducing the velocity.

PROFIsafe is a PROFINET-based safety interface that is used for communication between the PLC, the controller and the scanners. PROFINET is an Ethernet-based field bus, and is the industry technical standard for data communication over IWLAN [31, p. 21]. The system is characterized by the fast delivery of data in 1ms or less, which is considered real-time [32].

#### 3.1.4. LBR iiwa 14 R820

The LBR iiwa 14 R820, in Figure 3.4, is the manipulator that, together with the KMP, constitute the KMR iiwa. Sunrise Cabinet is the controller of the robot arm, and the electronics of the LBR is internally routed to the controller. The LBR is intended for handling tools and for processing or transferring components [33, p. 7]. The robot arm has a position accuracy of  $\pm 0.1$  mm, and can thus be used for precise assembly work. The designation 14 stands for the payload capacity of the robot, which is 14 kg. R820 refers to the maximum reach of the robot arm, which is 820 mm, as shown in Figure 3.5b.

The LBR is mainly made of aluminum and is classified as a lightweight robot. It is a jointed-arm robot with seven controllable axes, as shown in Figure 3.4, and is based on the structure of a human arm. Every joint has a motor for moving the joints and integrated sensors that provide information about position, torque, and temperature. The sensors in the axis have protective functions to control the



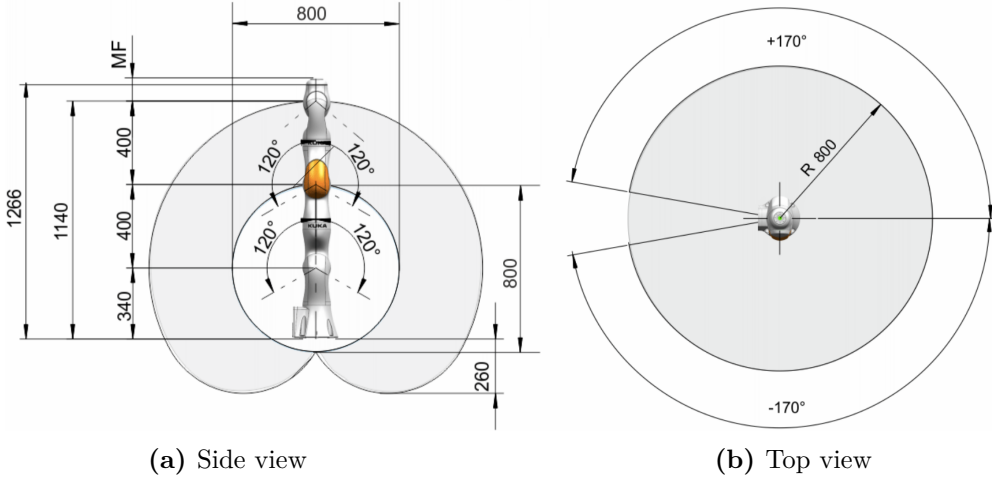
**Figure 3.4.:** LBR iiwa 14 R820 [34] [33, p. 10].

motion of the robot arm. Axis range sensors ensure that the axis range is adhered to, and torque sensors ensure that the permissible axis loads are not exceeded. Axis data for the LBR is listed in Table 3.3, including the motion ranges and the speed limitations for each axis. The workspace of the LBR is derived from the axis ranges. The workspace and the dimensions of the robot arm are showed in Figure 3.5.

**Table 3.3.:** LBR iiwa 14 R820 axis data [33, p. 20]

Axis	Motion range [deg]	Speed with maximum payload [deg/s]
A1	$\pm 170$	85
A2	$\pm 120$	85
A3	$\pm 170$	100
A4	$\pm 120$	75
A5	$\pm 170$	130
A6	$\pm 120$	135
A7	$\pm 175$	135

For Cartesian space motion commands, the controller needs to find the joint



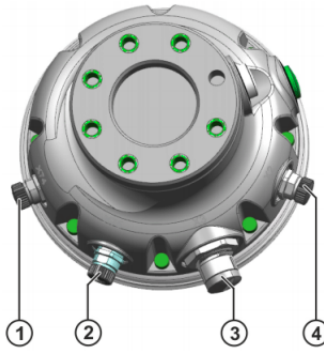
**Figure 3.5.:** Workspace of the LBR. Dimensions are in mm and degrees [33, p. 21].

positions along a Cartesian path and calculate the inverse position and velocity kinematics of the robot. Due to the axis positions and combinations of these, Cartesian motions of the robot may be limited. This is known as singularity positions. The mobility of the manipulator is reduced and should, therefore, be avoided. A typical case is if a small change in Cartesian space requires extensive changes to the axis angles.

### Media Flange

The media flange is an universal interface that makes it possible to connect components to the robot flange, such as tools or grippers. The media flange is integrated into the seventh axis of the LBR and is designated as *MF* in Figure 3.5a. The media flange is an universal interface that enables the user to connect electrical and pneumatic components to the robot flange. It provides transmission of data signals and comes in several variants with different interfaces. All media flanges have a hole pattern conforming to DIN ISO 9409-1-50-7-M6. The media flange used in this work is the Touch Electrical media flange, shown in Figure 3.6. This flange makes it possible to connect electrical components to the end-effector and contains four interfaces designated as circled numbers in Figure 3.6. ① (x74) is the interface for analog signals and CAT5, ② (x75) is external power supply, ③ (x3) is for internal power supply and digital I/Os and ④ (x2) is the EtherCAT connection port.





**Figure 3.6.:** Interface of the media flange Touch electrical [35, p. 147].

**Motion Programming**

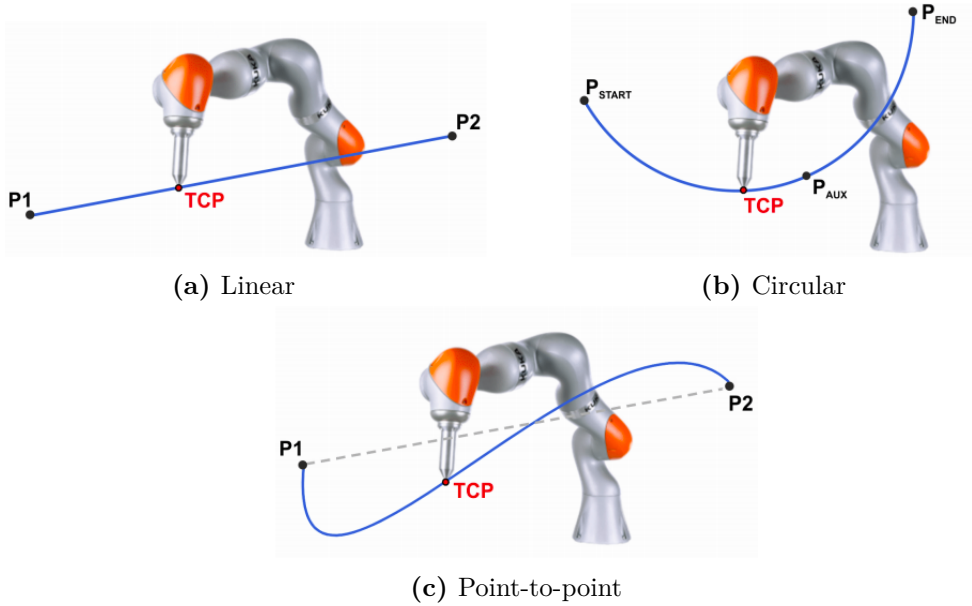
The LBR can be programmed by individual motions or by blocks of individual motions forming a trajectory. For all types of motion, the start point of the motion is the endpoint of the previous motion. The term Tool Center Point, TCP, is used for the reference point that is commanded to go to the requested position of a motion type. The TCP coordinate system is a Cartesian coordinate system and is, by default, equal to the flange coordinate system. The TCP should be configured as the working point of an eventual tool mounted at the flange.

Points in Cartesian space are defined as frames, where the point is the origin of the frame. Frames are defined by a transformation relative to its parent frame. The transformation describes how the frame is offset and oriented relative to a reference frame by the translations and rotations defined in Table 3.4.

**Table 3.4.:** Relative transformation between two frames

Translation [mm]	
<b>X</b>	Translation along the X axis from the origin of the reference system
<b>Y</b>	Translation along the Y axis from the origin of the reference system
<b>Z</b>	Translation along the Z axis from the origin of the reference system
Rotation [deg]	
<b>A</b>	Rotation about the Z axis of the reference system
<b>B</b>	Rotation about the Y axis of the reference system
<b>C</b>	Rotation about the X axis of the reference system

The available individual motions for the LBR are linear motion (LIN), circular motion (CRC) and point-to-point (PTP). These movements are illustrated in



**Figure 3.7.:** Individual motion types for the LBR [25, p. 374].

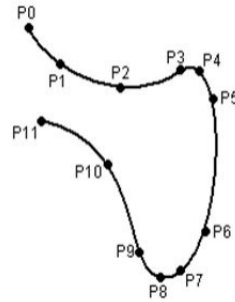
Figure 3.7.

Linear motion guides the TCP along a straight line in space to a specified endpoint. A circular motion guides the TCP along a circular path. An auxiliary point and an endpoint must be specified to calculate the circular motion. For both the LIN and CIRC motions types, the specified points are defined by a frame with absolute Cartesian coordinates.

A **PTP** motion guides the TCP along the fastest path to the defined endpoint. Due to the difference in maximum velocities for the joints, linear motion is not necessarily the fastest. The endpoint can be defined in the following ways:

- By specifying the angle configuration in radians for each of the seven joints.
- By defining the endpoint as a frame with absolute coordinates. If the end pose is defined in Cartesian coordinates, the joint configuration of the robot is not unique.

An individual motion can be programmed with motion parameters specifying the velocities, accelerations and jerk for each of the joints. The parameters can be defined as either absolute Cartesian values or as relative values in percentage of the axis-specific limitations. For a PTP motion, only the latter parameter option is available. The robot moves as fast as possible within the constraints of the physical limits and the programmed motion parameters. If no value is set for a



**Figure 3.8.:** A curved path with the Spline motion type, consisting of several PTP segments [25, p. 376].

parameter, the motion is executed with the fastest possible value.

There are two different types of grouped motions, Spline and Motion Batch. Common is that they consist of a group of individual motion segments that together form a path with the specified end poses of the individual segments. The difference is the approach used for approximate positioning. Approximate positioning allows a more smooth motion, as it does not go through the endpoint of each of the individual motions, but calculates the shortest motion approximately through the specified endpoints. Motion batch cuts the corner past a programmed point based on a defined approximation radius or distance. The robot never passes through the points, but point defines the contour of the motion. Spline motions are more computationally expensive, as they calculate a path to pass through each point rather than passing near them.

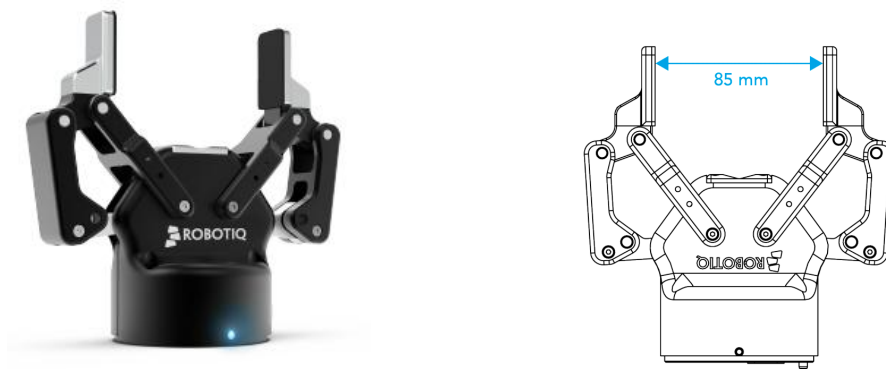
As illustrated in as Figure 3.8, Splines are suitable for complex paths, as the robot can execute curved paths in a continuous motion. A Spline can contain any number of segments of any length, as long as it is within the available memory. However, to reduce the calculation time of the path, it is recommended with a maximum of 500 segments per Spline, and that each segment is longer than 5 mm in a relative distance from the last endpoint [25]. Two different types of Splines exist, joint path (JP) and continuous path (CP). A JP Spline can only contain PTP motions, while a CP Spline can contain LIN and CIRC motion types. Each of the individual motions defines the velocities and accelerations for the motion. The velocity is reduced below the specified values for abrupt changes in direction or orientations and motions in the vicinity of singularities.

Motion commands can be executed either synchronously or asynchronously. Synchronous execution of a motion is done by the function `move()`. The motion commands are sent sequentially to the controller. The program is not further executed until the motion is finished. Asynchronous execution is done by the

function `moveAsync()`. The next command of the program is executed immediately after the motion command is sent. Approximate positioning is only carried out for motions executed asynchronously.

### 3.2. Robotiq 2F-85 Gripper

Robotiq produces universal robot grippers, sensors and controllers to make automation easy, fast and accessible. Robotiq 2F-85 Adaptive Gripper, shown in Figure 3.9, is an end effector designed for grasping and temporarily holding objects and is intended for installation on manipulators. It consists of two fingers with an opening of 85 mm in between, and a single actuator for opening and closing the fingers. Each finger has two joints and is under-actuated, meaning that there are fewer motors than the number of joints [36, p.11]. This allows the fingers to adapt to the shape objects to be grasped automatically and enables simple control of the gripper. Robotiq provides several options for the fingertips or finger pads, and they can easily be replaced. The 2F-85 gripper has embedded object detection using force sensors.



**Figure 3.9.:** The Robotiq 2F-85 Adaptive Gripper has two fingers and a opening of 85 mm [36].

A coupling must be used when mounting the gripper at the manipulator to integrate the electronics. The gripper is powered and controlled through the coupling via a device cable that carries a 24V DC supply and Modbus RTU communication over RS-485 [36, p.16]. Robotiq provides couplings with different bolt patterns to adapt the gripper to different manipulators. An adapter plate may be required if the available coupling is not compatible with the end effector at the robot.

Robotiq offers controllers that support other industrial communication protocols than the default Modbus RTU. The controller can be configured to support real-

time Ethernet protocols like Modbus TCP, EtherCAT and PROFINET. Each controller is by default configured for one protocol, but additional software can be downloaded from Robotiq Support to reconfigure the controller.

3.2.1. Gripper Register Mapping

The 2F-85 gripper has an internal memory that is shared with the Robotiq controller, illustrated in Figure 3.10.

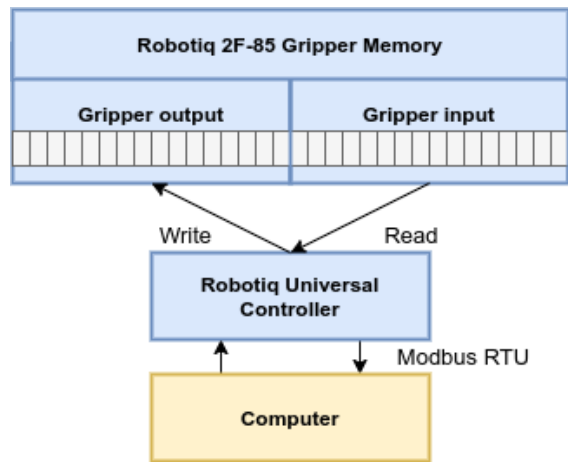


Figure 3.10.: Memory and control logic of the gripper.

One part of the memory is for robot output, and the other part of the memory is for robot input. Further, these are divided into registers defined in Table 3.5.

Table 3.5.: Registers of the 2F-85 Gripper [36, p.47]

Register	Gripper Input	Gripper Output
Byte 0	Action Request	Gripper Status
Byte 1	Reserved	Reserved
Byte 2	Reserved	Fault Status
Byte 3	Position Request	Position Request Echo
Byte 4	Speed	Position
Byte 5	Force	Current
Byte 6-15	Reserved	Reserved

The controller can write to the gripper input registers to activate functionalities or read the gripper output registers to retrieve the status of the gripper. Communication between the controller and the gripper is done through messages, indicating what registers to write or read. Each message consists of 16 bytes, where the hexadecimal numerical system encodes each byte.

## Gripper Input

The registers for gripper inputs with corresponding fields are listed in Table 3.6.

**Table 3.6.:** Inputs registers of the 2F-85 Gripper [36, p.51]

Robot Output	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Action Request	0		rARD	rATR	rGTO	0		rACT
Position Request	rPR							
Speed	rSP							
Force	rFR							

The Action Request stores information in four fields 3.6. **rACT** is the activate gripper bit, which enables activation of the gripper. Activation must be done before any other actions by setting the **rACT** bit to 1. **rGTO** is the bit corresponding to the Go To command. When this bit is set to 1, the gripper moves to the requested position according to the configuration set by the remaining registers. The bits **rARD** and **rATR** can be used for automatic release functionality for emergencies and is not intended to be used under normal operating conditions.

The bytes declaring the Position Request, Speed and Force each include one field. The hexadecimal *0x00* and *0xFF* correspond to the minimum and maximum values for the fields, respectively. The Position Request, **rPR**, is used to configure the position of the gripper's fingers. The minimum value corresponds to a fully opened position with 85 mm opening, while the maximum value correspond to fully closed. The Speed, **rSP**, is used to set the speed of the gripper movements. The Force, **rFR**, defines the gripping force. The gripping force decides the limitation of the current sent to the motor while in motion. If the current exceeds this limit, the gripper will stop, and object detection is triggered. Minimum force must be applied to very fragile or deformable objects, while maximum force can be applied for solid objects.

## Gripper Output

The registers for gripper outputs with corresponding fields are listed in Table 3.7.

**Table 3.7.:** Output registers of the 2F-85 gripper [36, p.48]

Robot Input	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Gripper Status	gOBJ		gSTA		gGTO	0		gACT
Fault Status	kFLT				gFLT			
Position Request Echo	gPR							
Position	gPO							

The status Gripper Status contains the fields **gOBJ**, **gSTA**, **gGTO** and **gACT**, as listed in Table 3.7. **gOBJ** is the Object Detection Status, which contain information about whether a grasp motion was successful. When picking an object with the **rGTO** command, the **gOBJ** bits provide information of the gripping motion as follows:

- Bits 00 = In motion: Gripper is in motion towards the requested position.
- Bits 01 = Object detection while opening: Gripper have stopped before the requested position was reached due to contact with an object while opening.
- Bits 10 = Object detection while closing: Gripper have stopped before the requested position was reached due to contact with object while closing.
- Bits 11 = Requested position: The gripper is at the requested position, and no object was detected.

Action Status, **gGTO**, echoes the defined **rGTO** bit, and is 1 if there is an active action request. **gSTA** provides information about the activation process, and the bits equal 11 when the activation is completed. The Fault Status returns error messages that are useful for troubleshooting. It can, for example, tell if the gripper is not activated before a Go To command is sent. Position Request Echo, **gPR**, echoes the requested position for the gripper and Position, **gPO**, provide the actual position read from the encoder.

3.2.2. Modbus RTU Communication

The gripper is controlled by the Modbus RTU serial protocol, which is a Master-Slave protocol [37, p. 5]. The slave in this context is the gripper. Each message is formatted according to the RTU Message Frame shown in Table 3.8.

Table 3.8.: RTU Message Frame [37, p. 13]

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0-252 bytes	2 bytes

The slave address for controlling the gripper using the Modbus RTU protocol is by default *0x0009* [36, p. 63].

The function code indicates what kind of action to perform. The following function codes are used in normal operations:

- Function code 16 (*0x10*) is used for sending functionality messages to the gripper.
- Function code 03 (*0x03*) is used for requesting information from the gripper.

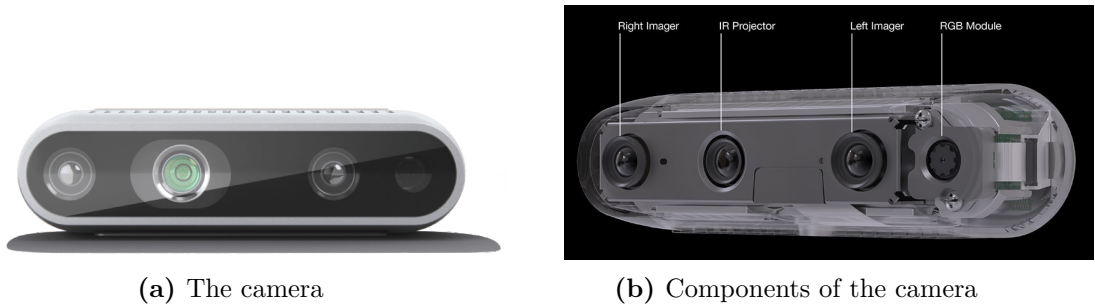
For each message with function code 16 received by the slave, the slave responds to the master by confirming that the functionality request is received. As slaves typically do not transmit data without a request, it is required to send a message with function code 03 to retrieve information from the gripper. Continuously sending messages with function code 03 can be used to retrieve the gripper output until either the requested position is reached or an object is detected. The Modbus RTU messages include a checksum generated based on the content of the message by the method Cyclical Redundancy Checking, CRC [37, p. 14].

3.3. Intel Realsense Depth Camera D435

Intel Realsense Technology is a series of technology from Intel which assist devices perceiving the environment. It consists of vision-based solutions like depth and tracking cameras, depth and tracking modules and lidars. There is also an open-source SDK that offers wrappers in multiple programming languages.

The Intel Realsense Depth Camera D435 is shown in Figure 3.11a. It is part of the D400 product line that was launched in January 2018. The camera uses stereo depth technology and a vision processor to perform depth sensing of the environment. The D435 camera has characteristics that make is a good fit for robotic applications. It is lightweight, has a small size, long range, real-time





**Figure 3.11.:** Intel Realsense Depth Camera D435 [38].

image processing and handles different lighting conditions [38]. A summary of the main specifications of the camera is listed in Table 3.9.

The D435 camera consists of an Intel RealSense Module D430, an Intel RealSense Vision Processor D4 and a RGB Camera. The depth module has two cameras, referred to as imagers and an infrared projector, which are used to calculate depth. The left and right imagers are identical cameras with identical settings [39, p. 15]. The infrared projector is used to project a random dot pattern onto a scene, which can further be used to discover low-texture surfaces. Typical low-texture surfaces are walls, doors and desks. The placement of the different components is shown in Figure 3.11b. The total depth FOV of the camera is  $87^\circ \pm 3^\circ \times 58^\circ \pm 1^\circ \times 95^\circ \pm 3^\circ$ , in respectively horizontal, vertical and diagonal direction. The field of view is wider than for other D400 camera models, which is advantageous in robotic applications to sense as much of the environment as possible at the same time. Another important feature in robotic applications is the real-time processing of images. This is made possible by the low powered Vision Processor D4.

The left and right imagers have a global shutter type, which is different from the rolling type of earlier camera models. A global shutter sensor scans the entire image simultaneously, instead of sequentially, line by line. The shutter makes the image freeze, and there is no blur on moving objects. This makes the camera a good fit for tasks like robot navigation or object detection, as both cases may involve motion. Another advantage of the global shutter is the low sensitivity to light, which makes it possible to navigate in an environment with low or bad lightning.

**Table 3.9.:** Summary of the main technical specifications of the D435 Camera [39, 38]

Specification	Value
Length $\times$ Depth $\times$ Height	90 mm $\times$ 25 mm $\times$ 25 mm
Weight	72 gr
Major components	Intel RealSense Module D430, RGB Camera, Intel RealSense Vision Processor D4
Image Sensor Technology	Global Shutter, $3\mu\text{ m} \times 3\mu\text{ m}$ pixel size
Depth Field of View (FOV)	Horizontal: $87^\circ \pm 3^\circ$ Vertical: $58^\circ \pm 1^\circ$ Depth: $95^\circ \pm 3^\circ$
Maximum range	Approximately 10 m
Minimum Depth Distance	0.105 m
Baseline	50 mm
Focal length	1.93 mm
Screw type	Camera mount screw (1/4"-20)

### 3.3.1. ROS Software

A ROS2 wrapper exists for using Intel RealSense D400 series cameras with ROS [40]. The camera node processes data from the camera sensors and publishes it to the appropriate topic making the data ready to use for ROS applications.

The published data and the respective topics are as follows:

- Rectified depth image: */camera/depth/image\_rect\_raw*
- Color image: */camera/color/image\_raw*
- Rectified infra1 image: */camera/infra1/image\_rect\_raw*
- Rectified infra2 image: */camera/infra2/image\_rect\_raw*
- Depth registered point cloud: */camera/aligned\_depth\_to\_color/color/points*

The rectified depth image is the processed stereo depth image coming from the imagers and the infrared projector. The color image is from the RGB Camera, and the rectified infra1 and infra2 images are from the left and right imagers. The depth point cloud is processed from the depth image. As the image data is taken with different cameras, it is defined in different frames. The camera node also publishes the transformations between the different frames.

**Table 3.10.:** Intrinsic parameters of the D435 Camera [41, p. 12]. All parameters exist for both left, right and RGB camera.

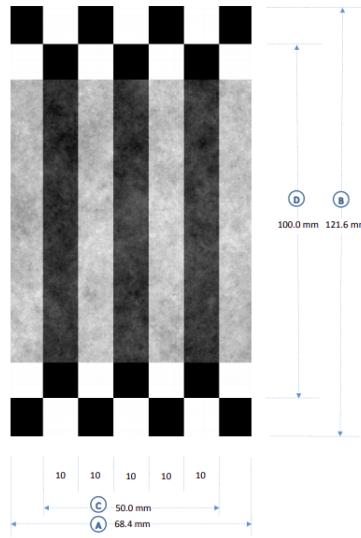
Parameter	Description	Specified as
Focal length	See Section 2.2.4	[fx; fy] in pixels
Principal point	See Section 2.2.4	[px; py] in pixels
Distortion	See Section 2.2.3	Brown's distortion model [k1; k2; p1; p2; k3]

**Table 3.11.:** Extrinsic parameters of the D435 Camera [41, p. 12]

Parameter	Description	Specified as
RotationLeftRight	Rotation from right camera coordinate system to left camera coordinate system	3x3 rotation matrix
TranslationLeftRight	Translation from right camera coordinate system to left camera coordinate system	3x1 vector [mm]
RotationLeftRGB	Rotation from RGB camera coordinate system to left camera coordinate system	3x3 rotation matrix
TranslationLeftRGB	Translation from RGB camera coordinate system to left camera coordinate system	3x1 vector [mm]

### 3.3.2. Calibration

Intel RealSense Technology provides calibration software of the cameras in the D400-series [41]. This software, called Dynamic Calibrator, makes it possible for the end-users to recalibrate their cameras easily. They also provide an API, Dynamic Calibration Tool API, which makes it possible for developers to create custom calibration applications. The software provides a dynamic calibration process where the motion of the camera is evaluated, which means only the extrinsic parameters are updated. It is assumed that this is a recalibration process, which means the nominal parameters are known. The intrinsic and extrinsic parameters of the camera are listed in Table 3.10 and 3.11. The left camera is the reference camera and is located at world origin.



**Figure 3.12.:** Calibration board for the Dynamic Calibrator [41, p.71].

The D400-series cameras support two different types of dynamic calibration, rectification calibration and depth scale calibration. These algorithms find the parameters necessary to rectify the images, as well as the extrinsic parameters. The DynamicCalibrator only support these algorithms in a targeted calibration mode. The target in Figure 3.12 can be printed or displayed on an Apple or Android phone through the app *DynamicTargetTool*. If the target is printed onto paper, the correct dimensions, shown in Figure 3.12, must be preserved.

The calibration procedure consists of both rectification and depth scale calibration. For devices with a RGB camera, as D435 has, a step for calibrating the RGB camera is also included.

# Chapter 4.

## ROS2

ROS is short for Robot Operating System and is an open-source operating system for robots. It includes tools and libraries to handle the software programming of robots without having to deal with hardware. A thorough introduction of ROS was made in the specialization project. The content is partly reproduced, as it is essential to have a basic understanding of ROS and how it works to understand this thesis. Of the reproduced content, some sections are shortened while others are extended based on the relevance of the thesis. Besides, new stacks and concepts are introduced.

Section [4.1](#) and [4.2](#) introduce ROS2 and important concepts. Section [4.3](#) introduces two SLAM stacks, Cartographer and RTAB-Map. Section [4.4](#) address the BehaviorTreeCPP library, which is not a ROS library, but a dependency for packages based on behavior trees. Section [4.5](#) and [4.6](#) describe Navigation2 and MoveIt2, which are used for path planning and controlling the mobile platform and manipulator. The libraries used for object detection are presented in Section [4.7](#).

### 4.1. Introduction to ROS

Even though ROS is called an operating system, it is a middleware. It is built on top of another operating system but provides all the services and functionality one would expect from an actual operating system. The middleware is handling the communication between different programs, and there exist plug and play libraries that can be used to customize the system for the desired application.

ROS2 aims to be more flexible and universal than the first version of ROS, and there is added support for real-time programming. ROS2 is built on top of DDS, Data Distribution Service, that provides a publish and subscribe transport concept similar to the one used by ROS. It offers a distributed discovery feature,

which allows any two DDS programs to discover and communicate with each other as in a peer-to-peer network [42]. The default ROS2 middleware interface is the DDS implementation Fast RTPS delivered by eProsima [43]. There exist several language-specific ROS2 client libraries based on the same functionality for ROS2. Rclpy and rclcpp based on Python and C++, respectively, are the two client libraries maintained by the ROS2 core team. Nodes written with different client libraries can communicate because all client libraries implement code generators to interact with the ROS interface [44]. ROS2 is state-of-the-art software and is currently under massive deployment. In general, ROS2 tries to be cleaner and faster, and it will most likely become the primary robot operative system in the future.

## 4.2. Concepts

This section briefly explain the most common used ROS terms and concepts.

A *node* is a process that performs computations. A robot system can typically be broken down into multiple separate parts, and each node handles one specific part. It could be to handle the localization, the navigation or a specific sensor. A goal is to build the nodes modular for easy reusability such that a node can be reused in a different project where the same purpose is needed. All the nodes are connected in a graph and communicate through topics and services. It can publish and subscribe to different topics, or provide or use services. For a mobile robot, a typical scenario could be that the node handling a laser scanner sends the laser data to the node handling obstacle detection. This node performs the necessary computations and sends this information to the node handling path planning.

A *parameter* is a configuration value of a node. A node can have an unlimited number of parameters, and the parameters only belong to that specific node. The parameters can be integers, floats, booleans, strings, and lists. They are usually set when creating the node but can be dynamically reconfigured using the communication concept ROS services.

The nodes exist within a structure called a *package*. All the software is organized in packages. The package includes everything needed to have a proper software module, such as programs, external dependencies, configuration files and data sets. The goal is to collect software in an easily reusable way. There exist thousands of official ROS packages in addition to packages created by users. Not all the official ROS1 packages are migrated to ROS2, but new ones are continuously ported. Packages with collective functionality can be organized into *stacks*.

The communication between nodes is based on sending and receiving *messages*. A message can be made of common data types like integers, floats and strings,

but also arrays or nested structures. Files describing the message type are called `msg` files and are simple files describing what kind of data structure this message type supports. A message can be of the standard data types, or a custom made message type put together by multiple fields of the standard types.

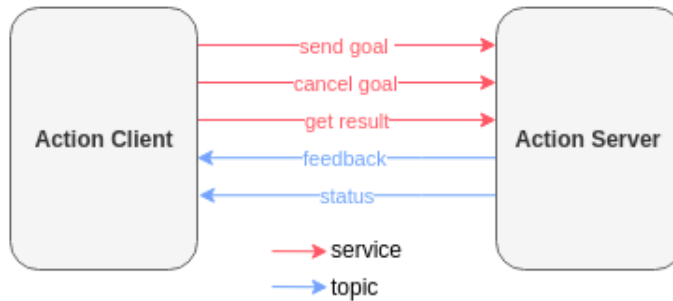
ROS utilizes three different methods for communication between the nodes, topics, services and actions. A *topic* works as a communication bus, based on the publish-subscribe pattern from software architecture. The topic is a middle-state that transfers messages between the nodes. The nodes that publish information, publishers, do not know which nodes that subscribe to the topic and keep publishing information regardless of whether there are any subscribers. The nodes interested in information, subscribers, express interest in a specific topic, and receives messages if there are any from the topic. They have no information about where the messages are published. A node can both be a publisher and subscriber at the same time, and there are no limitations in how many topics it can publish to or subscribe. A topic can only receive messages of one specific type, either a datatype or a ROS message type, which is defined when the topic is created.

Communication through topics never provides any feedback. If a node needs a response to a request, they should use *services* instead. ROS services are based on the client-server software architecture, where a client can send a message request to the server and receive a response. The service clients and servers are directly created inside the ROS nodes. A practical use case for a ROS service could be to reset a simulation and get a response to whether the request succeeded or not. A service works as a one-time communication, which means that the two nodes are disconnected when the response is returned.

ROS also provides a third way of communicating. *Actions* are used when a request takes a long time to complete, and there is a chance that the user wants to cancel the request or receive feedback on the progress. Action communication consists of two components: a server and a client. The action server provides an action while the client sends goals and monitors the feedback. Communication between the server and the client happens through multiple messages. These messages are defined in an action specification, `.action` file, which specifies the goal, result and feedback. The goal message is sent to specify the action, the feedback message is sent continuously and describes the progress of completing the goal, while the result message is sent exactly once when the action is completed.

Communication by actions is made up of three services and two topics. The services are `send goal`, `cancel goal`, `get result`, and the topics are `/feedback` and `/status`. This is illustrated in Figure 4.1. The `send goal` service provides the server with the goal, and the server returns whether the goal was accepted or rejected. The `cancel goal` service is used by the client if wanting to cancel a currently executed goal on the action server, and a response code indicating any

failures is returned. The get result service is used by the client to request the final result of the goal. The service returns the defined result message and the final status of the goal. The server publishes information on two topics. On the status topic, the current status of the goal is published. It can be either accepted, executing, canceling, succeeded, aborted or canceled. This topic is used by the server for introspection and is not used by the client. On the feedback topic, the defined feedback message is published by the server. An API for using ROS action methods is implemented in the client libraries, which means the users do not have to handle the topics and services directly.



**Figure 4.1.:** Middleware implementation of a ROS action, made up of three services and two topics [45].

The following paragraphs are not describing ROS concepts, but tools and software which are highly relevant when working with ROS.

A robotic system typically has many coordinate frames. *Tf* is a ROS package that keeps track of all the different frames and their relative transformations. Tf keeps track of the relationship between frames in a tree structure, and all frames of the robot need to be connected through a tf tree. The information is either published to the /tf\_static or /tf topic, based on if the transformation is static or dynamically changing. Each message includes a transform, both translation and rotation, from one frame to another.

*URDF* is an Universal Robotic Description Format used by ROS to describe the different parts of a robot and how they are connected. The URDF file is written in XML file format, and XML tags describe each component of the robot. The URDF describes all the characteristics of the robot in a human-readable way. The robot is described by using links describing the components and joints, which connect the components. The information related to the links can be visual features, inertia, mass and collision properties, describing the range of interference of the link. If the geometry of the link is simple, it can be described directly in the



URDF file. There is support for boxes, cylinders and spheres. If a more complex shape is required, it can be done by including a mesh. Each link has a separate frame, and the joints define the transformations between all frames. By using a ROS package called `robot_state_publisher`, the information from the URDF can be published to a topic and be available for other nodes depending on `tf`. Hence, the URDF should be an exact representation of the robot.

*Rviz* is a ROS visualization tool. It can visualize 3D models of a robot in addition to different types of data. The main purpose is to visualize the information from ROS topics, making it possible to validate if the data is correct.

*Gazebo* is a software used for simulation, modeling and testing of algorithms. It is a 3D simulator that provides models of robots, sensors and environments to perform realistic simulations. It has a physics engine that makes it possible to add physical constraints to the environment, which makes the simulations more realistic. Besides, different sensors and cameras are supported. *Gazebo* is open-source and has a high performance, which makes it a popular simulation tool.

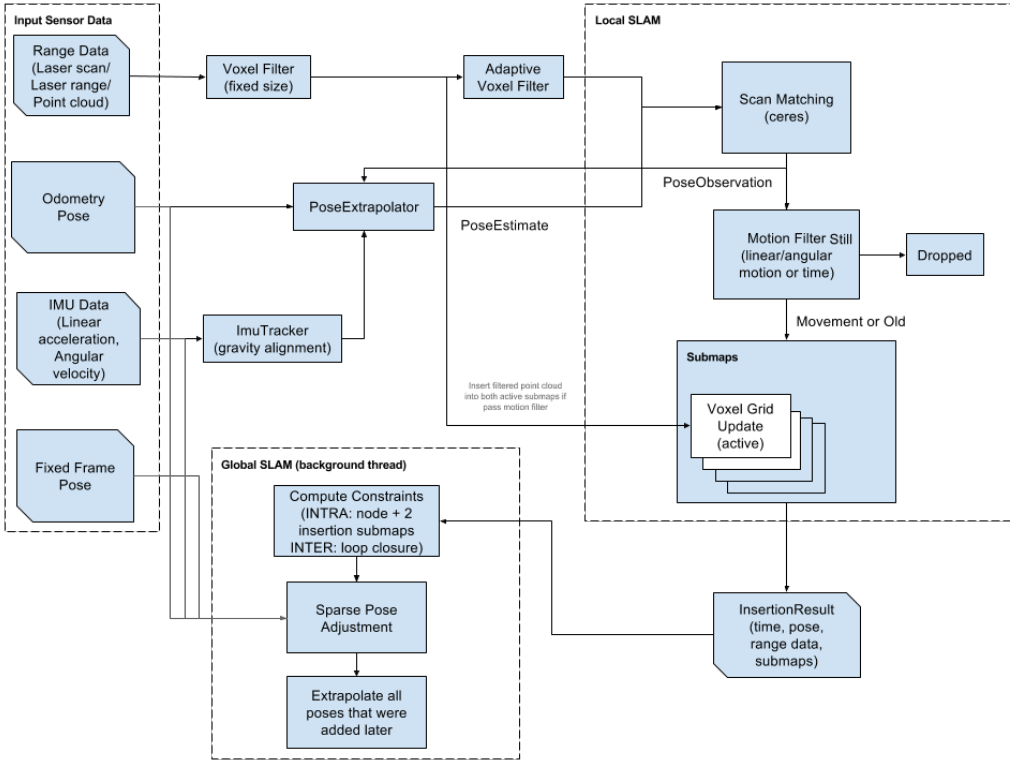
## 4.3. Stacks for SLAM

Two libraries that provide SLAM for ROS2 are Cartographer and RTAB-MAP. Common is that they support both lidar and visual SLAM and that they are based on a graph-based SLAM approach. Both stacks support 2D occupancy grid maps as output, which enable easy integration with navigation solutions such as Navigation2. The maps created can be saved by using the map server package included in Navigation2.

### 4.3.1. Cartographer

Cartographer is a part of Google's open-source code. It can be used for both 2D and 3D lidar graph-based SLAM, and the range measuring data can be of the types 2D scans, multi-echo sensor scans or 3D point clouds.

Figure 4.2 shows a block diagram of the Cartographer system. Cartographer is built on two subsystems for local SLAM and global SLAM, handled by the `cartographer_node` and `occupancy_grid_node`. The `cartographer_node` is the local trajectory builder that generates and publishes local occupancy grid maps. The `occupancy_grid_node` handles the assembly of submaps to a global map according to a S2M approach. Constraints between the submaps are added either as loop closures or as successive maps. The local SLAM approach is based on lidar odometry, but wheel odometry or data from an IMU can be provided to get more robust scan matching. The pose extrapolator uses sensor data to predict where



**Figure 4.2.:** The architecture of the Cartographer stack [46].

the next scan should be inserted into the submap. For both scan matching to local submaps and graph optimization, Cartographer uses Ceres Solver [47], which is an open-source C++ library for solving non-linear optimization problems. Finished submaps are added to the pose graph, which is optimized in the global SLAM thread running in the background. When the constraints graph is optimized, the submaps are rearranged to form a coherent global occupancy grid map.

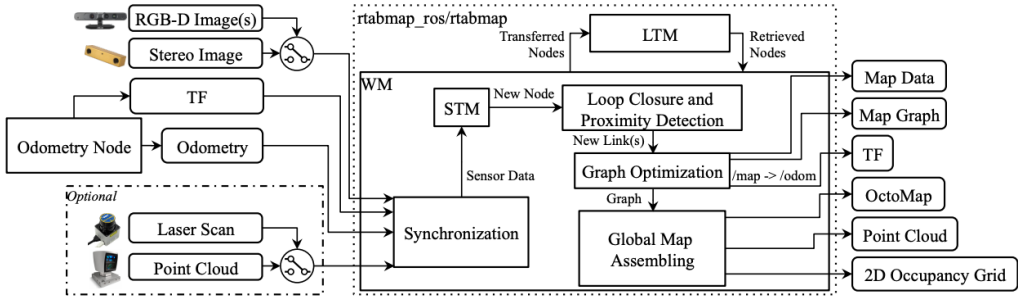
#### 4.3.2. RTAB-Map

RTAB-Map, Real-Time Appearance-Based Mapping, is an open-source library. A ROS wrapper exists to utilize RTAB-Map in ROS applications, which is maintained by Introlab [48]. RTAB-Map provides an approach for visual and lidar SLAM and implements a variety of solutions for input and output in both 2D and 3D. For sensor input, it supports depth, stereo and RGB-D images, and also laser scans and laser scan point clouds.

Figure 4.3 shows a block diagram of the RTAB-Map node in the ROS wrapper.

The required inputs are odometry and transformations to define the position of the sensors relative to the base of the robot. The odometry can be calculated based on odometry from the wheel encoders or odometry from measurement sensors. RTAB-map supports both [S2S/F2F](#) and [/F2M](#).

The input is synchronized before passed to the graph-based SLAM algorithm, using either exact or approximate synchronization. Exact synchronization requires that the input has the same timestamp, while approximate synchronization is used for data coming from different sensors, and attempts to synchronize the sensor topics with minimum delay.



**Figure 4.3.:** The architecture of the RTAB-Map stack [22].

The loop closure detector uses a bag-of-words approach to determine if a frame comes from a previously visited location. A new constraint is added to the graph if a loop closure is detected, then a graph optimizer minimizes the errors in the map. A memory management approach is used to limit the number of locations used for loop closure detection and graph optimization to enable real-time features for large-scale environments. The memory is separated into a short term memory, STM, a long term memory LTM and a working memory, WM.

## 4.4. Behaviortree.CPP

The library BehaviorTree.CPP [49] can be used to create and handle behavior trees. It provides features for creating behavior trees from XML-representations, running asynchronous actions and creating custom tree nodes as plugins that can be loaded at run-time.

Behaviortree.CPP uses behavior trees described by an XML file. The nodes in the tree can either be predefined nodes from the library or custom nodes. Custom nodes must be registered into a BehaviorTreeFactory before they can be used. All nodes are meant to be general and reusable and BehaviorTree.CPP uses ports

to have a dataflow within the tree. The ports can be used to provide input to a node and retrieve the output. The output from a node can also be used as input to another node. BehaviorTree.CPP provides the functionality blackboard, which is a place to store and share values between the nodes in the tree. The values are stored in the blackboard in key/value pairs. The input ports can read from the blackboard, and the output port writes to it. Ports are connected by using the same key for the output of one node and the input of another, such that they point to the same entry of the blackboard.

## 4.5. Navigation2

Navigation2 is a library with tools that can be combined to control mobile robots. The main goal of Navigation2 is to drive the robot from a start pose to a goal pose. This task can be split into subtasks, which are all handled by Navigation2. These include handling maps, localizing the robot, path planning, path following, obstacle avoidance and creation of costmaps of sensor data.

To navigate, information about the environment and about how the robot moves are required. Sensor data is used to create a local costmap, and the data can either be laser scans or point clouds. A static map of the environment, of the type occupancy grid map, is also required. Also, odometry data and transformations between the sensor frame and odometry frame must be available. The goal of navigation is to navigate to a given pose in the environment. The user can provide this pose either through topics or Rviz. The output from Navigation2 is continuously published velocity commands making the robot follow a planned path to the goal pose. These commands describe the linear and angular velocity in three directions.

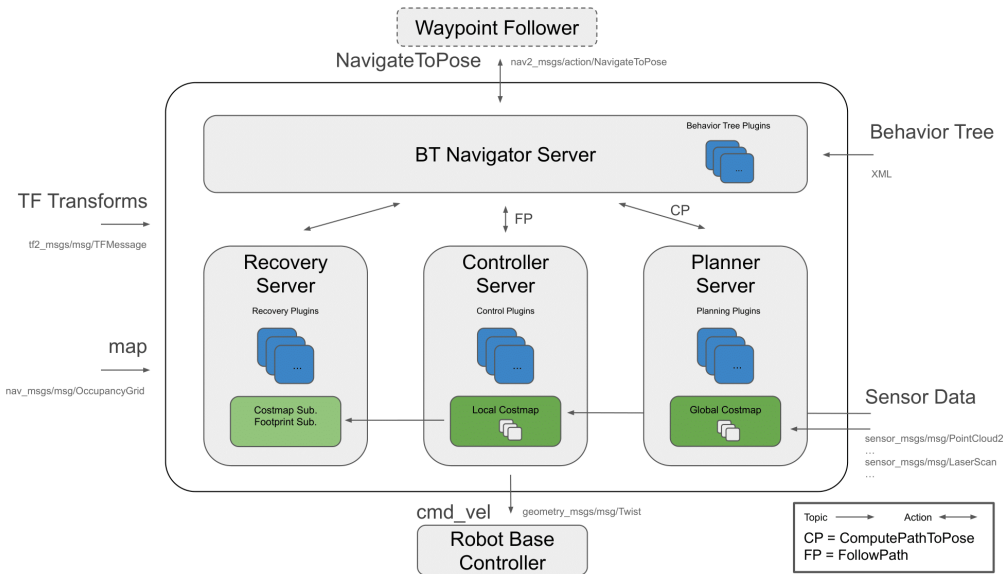
Navigation2 is built upon behavior trees that control the overall workflow. The BT Navigation Server handles this. The two primary action leaf nodes of the tree handle the computation of a path to a given pose and path following. The leaf nodes send requests to servers to perform the actions. These servers can be replaced with various algorithms performing the same action.

The global path planner, Navfn, performs the creation of a path. Navfn implements the Dijkstra search algorithm and uses a global costmap to find the path with a minimum cost between the start and end points. Dijkstra algorithm is guaranteed to find the shortest path under any condition. The local planner handles the path following. The local planner, called DWB, uses the plan from the global planner and a local costmap to produce the velocities making the robot reach the goal state. DWB is an improvement of the local planning algorithm, [DWA](#), applied in ROS1.

While navigating, the robot is also localized on the map. The localization algorithm [AMCL](#) performs the localization. The key idea of AMCL is to express a hypothesis about the robot's pose by a set of weighted particles. The sensor readings are used to give each particle a weight based on how likely it is that the particle represents the correct position of the robot. Particles with low weight are unlikely and will be replaced with more likely particles, making the particles converge to the correct pose of the robot. Random particles are inserted to be able to recover from false convergence.

The Navigation2 stack includes recovery behavior, which is triggered if the robot is lost. Currently, this includes performing spins in-place, performing a linear translation by a given distance, and stopping, which brings the robot to a stationary state.

Figure 4.4 shows the architecture of the system. As Navigation2 is under heavy development, the different parts of the architecture have been changing rapidly, and some are still subject to changes.



**Figure 4.4.:** The architecture of the Navigation2 stack [50].

## Local Planner

The basic idea of the DWA algorithm, which the local planner DWB is based on, is to find the available velocity commands and evaluate how they would perform. A velocity search space is created based on a discrete sampling of the control space,  $dx$ ,  $dy$  and  $d\theta$ , to find out what command velocities are available. Simulation of

each sampling is performed to see what would happen if the velocity were applied for a short time.

Different critics then evaluate these trajectories. Each critic gives the trajectories a score based on specified criteria before the scores are combined to give a final value for the trajectory. The trajectory with the highest score is chosen and determines the velocities of the robot. Each of the critics is given a weight based on the importance of the criteria. The weights can be tuned to control the qualities of the chosen trajectories.

The following critics are available in Navigation2:

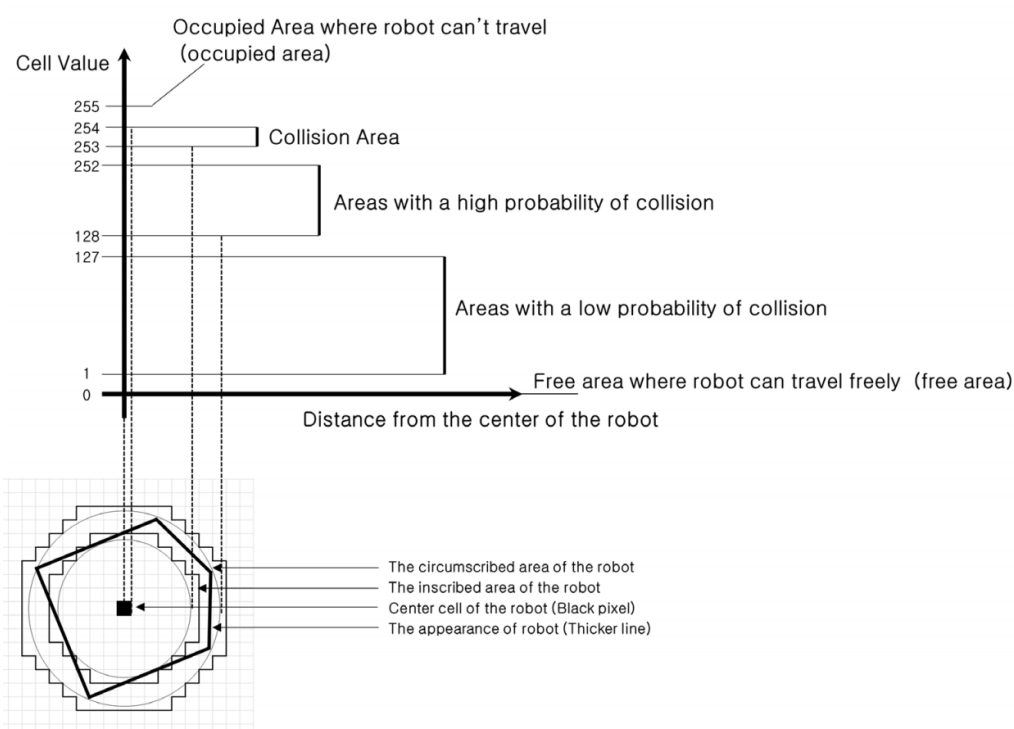
- **BaseObstacle:** Gives a score based on whether the path is in collision at any of the poses in the trajectory.
- **ObstacleFootprint:** Gives a score based on whether the robot's footprint touches obstacles or not. Only the border of the footprint is evaluated.
- **GoalAlign:** Gives a score based on how well the trajectory is aligned with the goal pose.
- **GoalDist:** Gives a score based on the distance from the trajectory to the goal pose.
- **PathAlign:** Gives a score based on how well the trajectory is aligned with the path created by the global planner.
- **PathDist:** Gives a score based on how far the trajectory is from the global path.
- **PreferForward:** Gives a score based on how much the robot moves forward. Trajectories with backward movement and turning are penalized.
- **RotateToGoal:** Penalizes trajectories consisting of linear movements, if within a certain distance from the goal.
- **Oscillation:** Gives a score based on if the trajectory is oscillating.
- **Twirling:** Penalizes trajectories with pure rotational velocities.

## Costmaps

To handle path planning and obstacle avoidance, Navigation2 relies on costmaps, which is a dynamic representation of the space. In a costmap, each grid is categorized based on if the area could be moved to or not, and the grid values are costs. Each pixel has a value between 0 and 255. The range is split into different

classes: free, low collision probability, high collision probability, collision area or occupied.

Navigation2 is based on a layered costmap structure where different costmap layers can be defined, constituting one full costmap. The available costmap layers are static, obstacle, voxel and inflation layer. The static layer is a representation of the occupancy grid map. The obstacle and voxel layers are both based on sensor data. While the obstacle layer is created based on 2D data, the voxel layer uses 3D data. The inflation layer is where the obstacles are inflated to define a cost for the surrounding cells. The cost values are strongly related to the distance between the robot and the obstacle, and the relationship is shown in Figure 4.5.



**Figure 4.5.:** Costmap cell value seen in the context of the distance to the obstacle [51, p. 356].

Navigation2 uses both a global costmap and a local costmap. The global costmap inflates the obstacles in the occupancy grid map and is used for creating a long-term path. The local costmap is not related to the static map and is built by inflating obstacles observed by the robot's sensors. This happens in real-time and is used for local planning.

## 4.6. MoveIt2

The motion planning problem for manipulators consists of similar challenges as for a mobile robot. The goal is to find a path between a start position and an end position, which avoids collisions and is cost-optimized. MoveIt is one of the most used manipulation software for ROS and is used by more than 100 different robots [52]. MoveIt2 Beta is the second version of MoveIt, created for ROS2, and was launched in February 2020. It is the first beta version released for MoveIt2. It has most of the core functionality from MoveIt, including motion planning, collision checking, trajectory execution, inverse kinematics and visualization in Rviz [53]. MoveIt2 plans to utilize the advantages of ROS2 to enable faster and more reactive planning and realtime control of robot arms [53]. MoveIt has a higher level ROS interface, MoveGroup, and a graphical interface, MoveItSetupAssistant, which are meant to be a user-friendly access to the functionality. This feature is not yet ported to MoveIt2, which means users will have to use the APIs directly. A higher-level interface for MoveIt2 called MoveItCpp does exist, which does not use ROS functionality to access the core functionality. As many ROS stacks, MoveIt is based on a plugin architecture, where algorithms and libraries can be changed with a better fit.

For simplicity, MoveIt2 will hereby be referred to as MoveIt. If the information is specific for either the first or second version of MoveIt, they will be referred to as MoveIt1 and MoveIt2.

### 4.6.1. Configurations

MoveIt uses different plugins for almost every task, and the plugins can be configured using ROS parameters. Each plugin has a separate configuration file with parameters. Two files for describing the robot are the URDF and SRDF files.

#### URDF

MoveIt uses the URDF file to get a description of the robot and information for motion planning. The *limit* tag specifies the limits of what the joint can handle regarding joint movement, effort and velocity. The URDF also specifies a visual and collision mesh. The collision meshes are included in the collision checking performed during the planning. These should cover the relevant area, but at the same time not be too detailed. An over-detailed mesh deeply affects the collision checking time.



## SRDF

Another file describing the robot is the SRDF, Semantic Robot Description Format, file. It complements the URDF file and specifies robot configurations and additional information necessary for the planning.

Groups are a concept from the SRDF file, which is a collection of links or joints. The child links of a joint or the parent joint of a link are automatically included. Actions performed by MoveIt are done on a group, and a motion plan is created for a specific group.

Some joints are unactuated, and not possible to control, even though they are not fixed. These joints should be specified as passive joints in the SRDF to make sure the planners know they cannot plan for this joint. An example of a passive joint is a caster. Another type of joint that should be specified is virtual joints. A virtual joint can be used to specify the relationship between the robot and the world; hence, it defines the pose of the robot relative to the world. For a manipulator, this joint is typically fixed, but for a mobile robot moving around in the world, it could be a planar joint.

The SRDF should also be used to define information related to collisions between the links. The collision checking process checks by default for collisions between all joints. Pairwise collision checking between links can be disabled. This should be done between adjacent links, which are already in a collision, and between links where a collision can never occur.

Configurations of the joints that are used regularly can be defined and saved with a unique id, which makes it possible to reuse the pose easily. A configuration can be created as a group state, where id, group, and value for each of the joints are specified. This could, for example, be the default position of the manipulator.

### 4.6.2. How MoveIt Works

As mentioned, a motion planner is specified through a plugin. The most popular planner in MoveIt is Open Motion Planning Library [54], OMPL, and it is currently the only planner supported for MoveIt2. OMPL is open-source and implements randomized sampling-based planning algorithms [55].

A motion plan request specifies what the planner should do, and includes constraints and a goal for the planning. The different constraints can be:

- The joint, velocity and acceleration constraints specified in the URDF.
- A position constraint, where the position of a link must be within a region.

- An orientation constraint, where the orientation of a link must be within specific limits.
- A visibility constraint, where a point on a link must be within the visibility of a sensor.

A goal for the plan can be set in four different ways:

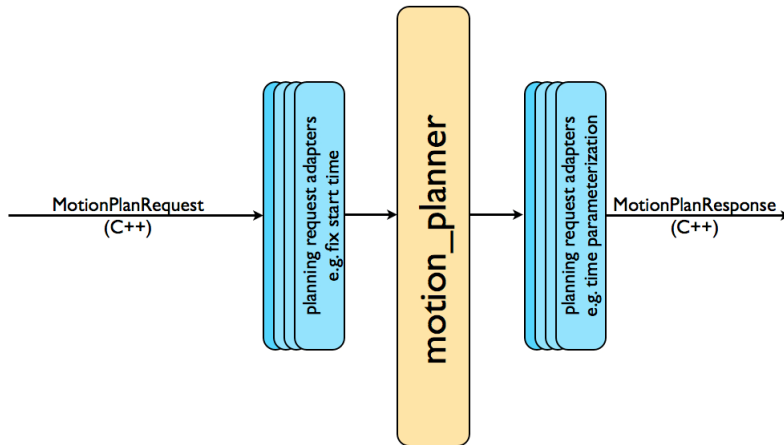
- By giving the name of a pose configuration specified in the SRDF file.
- By giving a pose and a link, where the link should end in this pose.
- By giving a RobotState object. A RobotState is a representation of a robot's state, which includes position, velocity, acceleration and effort.
- By defining a MoveIt Constraint message. A Constraint message includes four lists defining joint constraints, position constraints, orientation constraints and visibility constraints for each joint.

Parameters defining how the planner should look for a trajectory can be defined in the motion plan request. Examples of parameters are the number of planning attempts, allowed planning time and a scaling factor of the max velocity and acceleration.

When the motion plan request is processed and sent to the planner, a motion plan result is returned. A planner typically generates a plan, which means there is no timing involved between the points in the plan. MoveIt includes a trajectory processing routine, which handles the plan and generates a trajectory. A trajectory is both a path and information about how to traverse the path with respect to time. The plan is time parameterized by adding the constraints for joints, velocity and acceleration. This makes sure every waypoint is reached within a fixed time interval.

The process of creating a motion plan response from a motion plan request consists of a pipeline with multiple stages. It must include the motion planner, but can also include multiple stages for preprocessing of the motion plan request, and post-processing of the motion plan response. These stages are named planning request adapters. The motion planning pipeline is represented in Figure 4.6. The mentioned time parameterization is such a post-processing stage. An example of a preprocessing stage could be to fix the start state bounds. This stage makes sure the start state of the joints are within the joint limits, which may be necessary if the joint limits are not properly configured. When the motion plan response is completed, the created trajectory will move the group of joints to the desired location.

By default, collision checking is performed during planning, and MoveIt uses the open-source library FCL, Flexible Collision Library [57]. The collision checking



**Figure 4.6.:** The MoveIt motion planning pipeline [56].

is performed on both meshes, primitive shapes and octomap objects. Octomap is a library implementing 3D occupancy grid mapping approaches to create maps representing a full 3D model of the world [58]. During motion planning, collision checking is the most expensive operation, and the number of collision checks should be reduced if possible [56]. This is why it is helpful to disable collisions that never occurs in the [SRDF](#) file.

MoveIt also uses a plugin for handling kinematics of the robot. The default inverse kinematics plugin is KDL, Kinematics and Dynamics Library [59].

A planning scene represents the environment. It represents the world around the robot, and also the state of the robot. A planning scene monitor manages a planning scene. The planning scene monitor listens to state information from the robot concerning the joints, sensor information and user input. The planning scene is used both during planning and collision checking.

## 4.7. Stacks for Object Detection

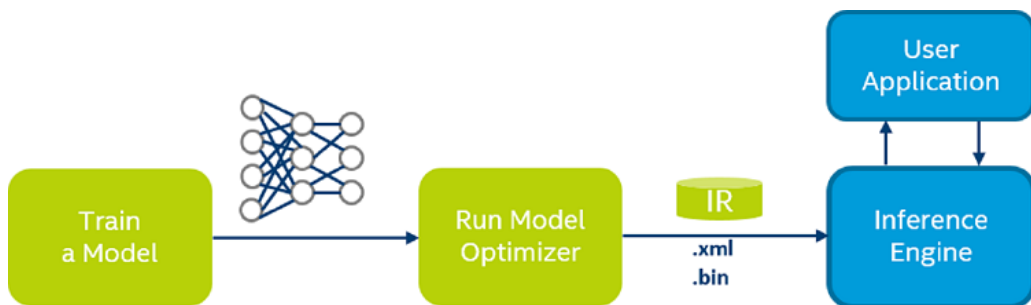
Intel Robot DevKit Project contains robotics-related open-source software components under the ROS2 framework. It can be used for perceptual computation, neuron network-based object detection and 3D localization with RealSense data [60]. This section presents two ROS2 stacks from this project and how they can be applied together to obtain 3D localization of objects.

The ROS2 OpenVINO toolkit is based on the general OpenVINO toolkit, which is first presented to be able to describe the ROS2 wrapper further. The stack

can be used for detecting objects in 2D images by inferencing data through a neural network. Another stack is Object Analytics, which consumes data from the camera and the detected object to localize the objects in 3D.

#### 4.7.1. OpenVINO Toolkit

Open Visual Inference and Neural Network Optimization, OpenVINO, is a toolkit developed by Intel. The toolkit is based on convolutional neural networks and extends computer vision workloads across, mainly, Intel hardware. OpenVINO includes different libraries and tools, such as DLDT, OpenCV and Open Model Zoo. The Deep Learning Deployment Toolkit, DLDT, provides deep learning implementation and consists of the Model Optimizer and the inference engine to run deep learning networks [61]. Model Optimizer is used for training networks, and include functionality to import pre-trained models from deep learning frameworks and convert it to an optimized unified intermediate representation, IR, file [62]. The IR graph is represented by a xml file and a binary file. The XML file describes the network topology by its layers and edges and defines the mathematical operations for each node, while the binary file contains information about weights and biases.



**Figure 4.7.:** OpenVINO Toolkit workflow for deploying a deep learning model [62].

The IR network can be read and inferred with the inference engine, as illustrated in Figure 4.7. The inference engine is a C++ library that allows high performance on the inference of images and integration into customized applications.

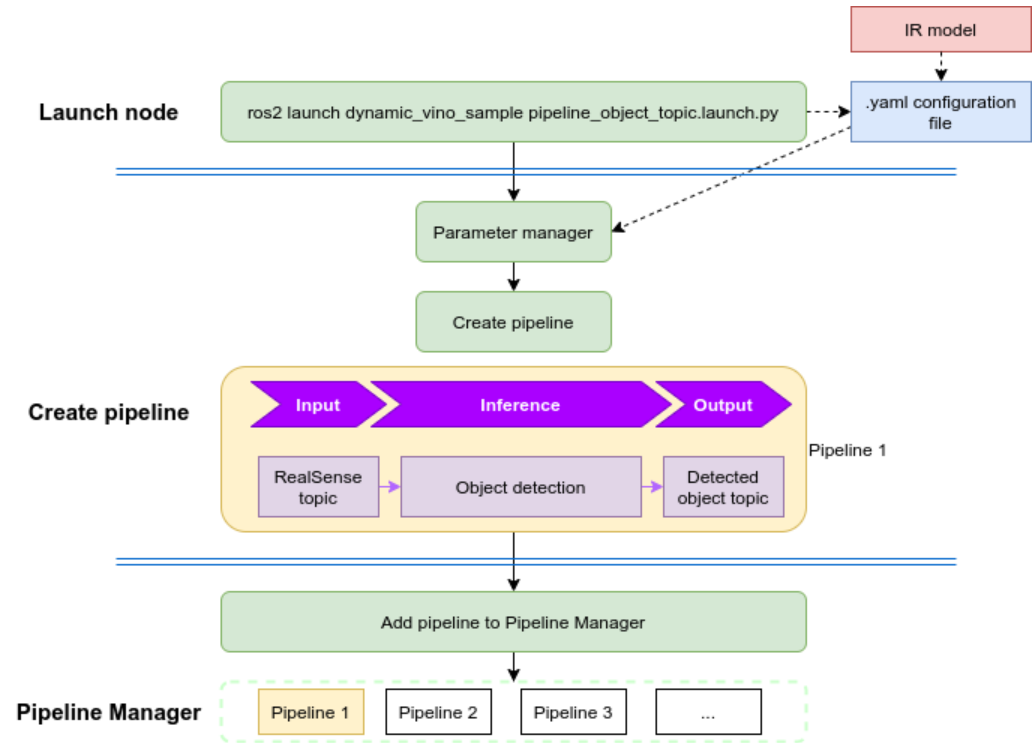
Open Model Zoo is a repository with ready-to-use, public deep learning networks that are trained by Intel teams. The pre-trained models can be used instead of generating custom networks to speed-up the development and production deployment process.

OpenCV is an open-source library developed by Intel that includes computer

vision algorithms and machine learning and deep learning functionality. OpenCV supports models from several deep learning frameworks like TensorFlow.

**ROS2 Openvino Toolkit**

The ROS2 wrapper extends the OpenVINO toolkit and adapts it to the ROS system. The inference engine supports different types of detection, whereas object detection will be further addressed as it is the feature of interest. Intel Realsense cameras are supported as input for gaining image data, which can be provided via ROS topics. The stack includes the entities pipeline, parameter manager and pipeline manager. Figure 4.8 illustrates how the components co-work together for object detection.



**Figure 4.8.:** Logic flow for creating pipelines with the OpenVINO stack.

The pipeline node is a default for the different features and must be launched with a specified configuration file, including an IR model. The parameter manager processes the configurations for the pipeline and passes the parsed configuration to the pipeline procedure. A pipeline instance is created according to the configurations and further included in the pipeline manager for lifecycle control and

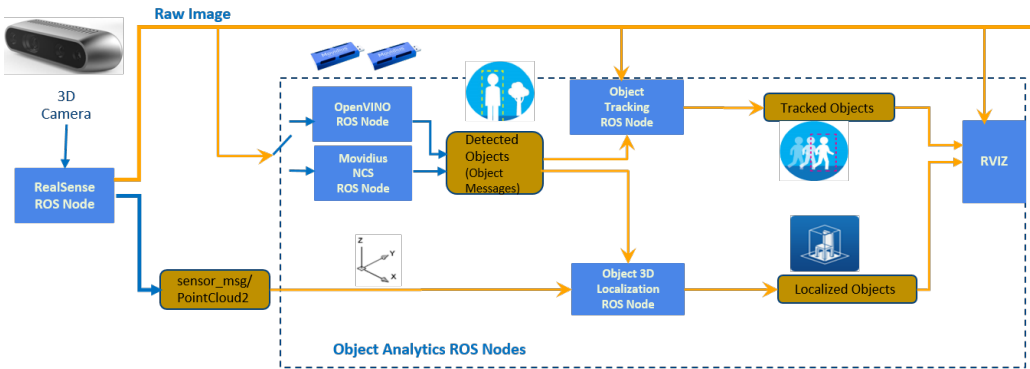
inference action triggering.

The pipeline processes the data from the resource component, builds up the structured inference network, and passes the data through the network. The inference results are further published to a topic containing messages of the type `Object-InBox`. This message includes information about the object type, the detection probability and the region of interest in the image. The region of interest is defined as a 2D bounding box in pixels that encloses the object in the image.

The pipeline manager manages all the created pipelines and makes it possible to run more than one pipeline in the same process. This way, multiple pipelines with the same resource can be created with custom configurations. The pipeline manager also has features for inference requests or external demands from the user. This is implemented by the use of a ROS service, which can be used for starting, stopping, pausing and retrieving status information from existing pipelines.

#### 4.7.2. Object Analytics

Object Analytics is a ROS2 stack for real-time object tracking and localization in 3D [63]. The object detection pipeline is shown in Figure 4.9. Only the localization task will be addressed, as it is the feature of interest for this project.



**Figure 4.9.:** Architecture of the Object Analytics pipeline [63].

The object analytics node takes detected objects from the OpenVINO node and data from a camera as input and provides the 3D pose of the objects in the image. More specifically, the node subscribes to the topics containing color images and point clouds from the Realsense node. The node processes the image data from the camera and localizes the detected objects in the camera frames in 3D bounding boxes. The localization is done according to the approach described in Section 2.2.8, which is based on the Organized Multi Plane Segmenter Algorithm.

The object analytics node publishes customized ROS messages of type `ObjectsInBoxes3D`. Such a message contains information about the detected object, the region of interest in the image and the bounding box enclosing the object in 3D. The bounding box is defined by minimum and maximum values that locate the diagonal of the box in the camera coordinate frame.





## Part II.

# Achievements and Evaluation



# Chapter 5.

## System Description

This thesis aims to create a concept towards a fully autonomous system focusing on fetch and carry operations as an important representative for industrial tasks. This includes utilizing the KMP and range sensors for navigation, the LBR and the Robotiq gripper for fetching objects and Intel Realsense cameras for detecting objects and sensing the environment.

Setup and preparations for the system are presented in Section 5.1. Section 5.2 presents the models created for describing the robot and using it in simulation. An overall overview of the physical architecture is presented in Section 5.3. Three PCs control the system, a control PC inside the KMR, an Intel NUC and a remote PC. The two latter both have ROS Eloquent installed, and are further referred to as external PCs. The communication between the control PC and the remote PC is described in Section 5.4. Section 5.5 concerns the software on the control PC. The software running on the external PCs is primarily controlled by a behavior tree. This implementation is described in Section 5.6. The behavior tree further communicates with multiple server nodes, which are connected to external hardware and software. The server nodes are described in Section 5.7.

### 5.1. Setup

For the system to function, setup and preparations of the different components are conducted. This applies to both physical components, described in Section 5.1.1, and software setup, described in Section 5.1.2.

#### 5.1.1. Physical Installations

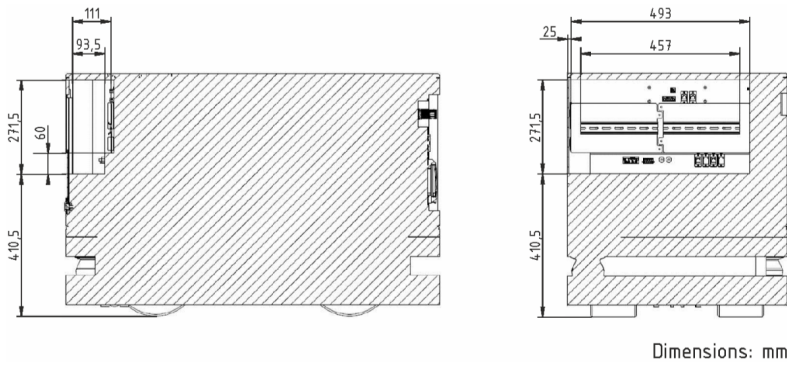
In addition to the KMR, the physical components in the system include a small form factor PC, Intel NUC, four Intel Realsense D435 depth cameras and a Robo-

tiq 2F-85 gripper.

### Intel NUC

An Intel NUC mini PC is used as an onboard computer to provide electricity and function as a communication medium to the external components installed on the robot.

The KMP has an installation space for additional components in the front of the vehicle, shown in Figure 5.1. This space includes different interfaces for connecting to the robot. The NUC is stored inside this space.



**Figure 5.1.:** Installation space for additional components in the KMP [23, p. 45].

The computer supports an input voltage of 19V. As the voltage outputs on the robot only provide 48V DC or 24V DC, an DC-AC power inverter is used to provide the NUC with electricity. This inverter uses the 48V DC power interface in the installation space of the KMP.

Ubuntu 18.04 and ROS Eloquent are installed on the NUC, making it possible to communicate with other ROS nodes through Fast RTPS and the ROS network.

### Robotiq 2F-85 Gripper

The default gripper coupling is not compatible with the media flange of the LBR, and an external adapter is used. There is no adapter that fits this exact combination at the retailer, so this had to be created. It was desired to attach a camera to the manipulator, and hence, a camera mount had to be included on the adapter. The adapter was 3D modeled and machined in the laboratory at NTNU. The finished adapter can be seen in Figure 5.2.



**Figure 5.2.:** The adapter used to connect the gripper to the media flange of the LBR.

The gripper is connected to a Robotiq universal controller which is supplied power by the 24V DC interface in the installation space of the KMP. The controller is connected to the NUC by USB, making it possible to communicate with the gripper over Modbus RTU.

### Intel Realsense D435

Four Intel Realsense D435 cameras are used in this setup. Three cameras are installed at the mobile base to use within SLAM and navigation. The last camera is installed at the manipulator to perform object localization. The D435 cameras use a different type of screw than the mounting holes at the mobile base. To be able to attach the cameras to the base, three mounting plates were created. The metal plates are rectangular with three screw holes. Two of them attach the plate to the mobile base, while the last one is used to attach the camera to the plate. One camera is placed at the front of the base, one at the left side and one at the right side. Each camera has a horizontal field of view of  $87^\circ \pm 3^\circ$ , which means the fields are not overlapping. The camera at the manipulator is attached to the created adapter. The cameras are connected to the NUC through USB. Figure 5.3 shows the camera at the manipulator, while Figure 5.4 shows the cameras attached to the mobile base.



**Figure 5.3.:** A D435 camera is mounted at the end-effector.



(a) One camera installed on each side



(b) Camera installed at the front

**Figure 5.4.:** Three D435 cameras mounted on the KMP.

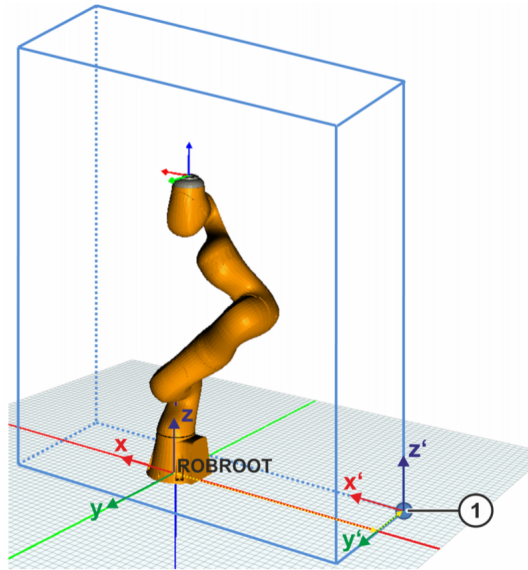
### 5.1.2. Software Setup

The software setup includes a description of safety configurations of the KMR required to make the robot compatible with our system, followed by a description of how the object detection model was created.

## Sunrise Safety Configurations

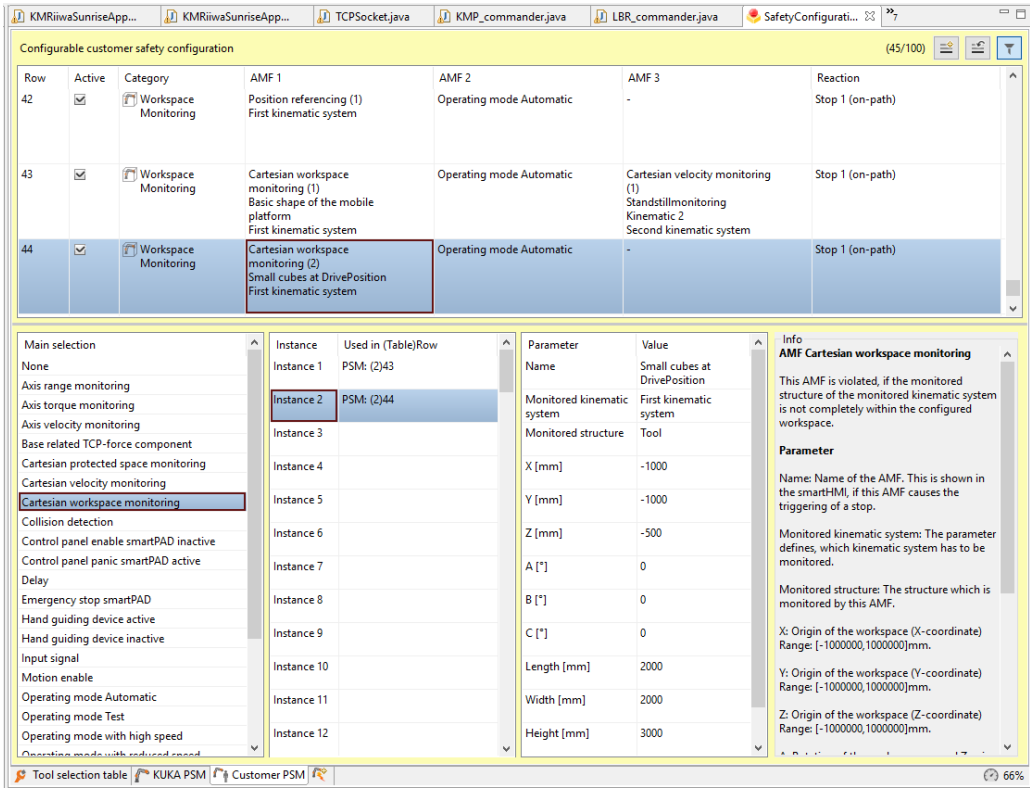
The safety configuration of a Sunrise project contains tables defining different restrictions with corresponding consequences. The Customer PSM (Permanent safety-oriented monitoring) table includes user-specific safety functions. The safety functions are preconfigured by KUKA, but can be deactivated and changed by users.

The row *Small cubes at DrivePosition* specifies a monitored workspace. When used in automatic operating mode, the LBR must remain inside the workspace at all times. The workspace is defined as a Cartesian cuboid relative to the manipulator's coordinate system. An example of such a workspace is illustrated in Figure 5.5, where ① denotes the origin.



**Figure 5.5.:** Example of a cartesian workspace for the LBR. ① denotes the origin of the workspace [25, p. 298].

The predefined size of the cuboid only allows small perturbations around the drive position. If the LBR moves outside the cuboid, an emergency stop is triggered. This safety configuration is included to avoid collisions. The size of the space was increased, such that the robot arm could be fully stretched in every direction.



**Figure 5.6.:** Safety configuration Customer PSM table in Sunrise Workbench. The row *Small cubes at DrivePosition* is adjusted to increase the workspace of the LBR.

Figure 5.6 shows the Customer PSM table and the row that was changed. The new workspace is defined as a cuboid with size  $2m \times 2m \times 3.5m$ .

### Object Detection Model

This section presents the process of creating an object detector model for the circular box shown in Figure 5.7.

A dataset was created based on approximately 1000 images of the circular box. The images were created as snapshots from videos capturing the box, taken with varying points of view, scale and light adjustments.

LabelImg [64] is a graphical image annotation tool that can be used to label objects in bounding boxes in images. Each image was labeled through LabelImg, which creates an associated XML annotations file. The annotation file includes information about the image in addition to the label name and the specified





**Figure 5.7.:** The black circular box used for object detection and grasping throughout the project. The box has a diameter of 7 cm.

coordinates of the bounding box. Thereafter, the dataset was split into a training set and a testing set. The dataset was divided according to the Pareto principle, namely 80 % of the images in the training set and the remaining as the testing set.

TensorFlow [65] is an open-source framework developed by Google designed to build neural networks for machine learning. TensorFlow Object Detection API [66] is a framework built on top of TensorFlow containing tools and pretrained models to construct customized object detection models. A model of the data set was created by utilizing the TensorFlow Object Detection API. First, a label map was created, which maps the labels of the data set to an id. This file is further used for training and detection processes. TensorFlow Records were created for each subset by running converting scripts<sup>1</sup>. TensorFlow Records are TensorFlows’s custom binary storage format which are the required input data for the training process.

A pretrained model from TensorFlow [67] was used to initialize the model. This concept is referred to as transfer learning, where a model developed for a task is reused as the starting point for a model for a different task. The model *ssd\_inception\_v2\_coco* was chosen as it provides a good trade-off between performance and speed [68]. Based on these files and the TensorFlow records, the model was retrained and evaluated by a script<sup>2</sup> from TensorFlow Object Detection API. The model is trained until it reached satisfying metrics.

TensorBoard is a program that can be used to monitor and evaluate the progress of the training. The model is evaluated based on metrics as precision, recall and loss.

---

<sup>1</sup><https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html>

<sup>2</sup>[https://github.com/tensorflow/models/blob/master/research/object\\_detection/model\\_main.py](https://github.com/tensorflow/models/blob/master/research/object_detection/model_main.py)

The training was terminated at step 30,661, which corresponds to a period of approximately 6 hours. The snapshots were taken at half-way through, approximately step 14,000, and at the end of the training process. Each of the rightmost subimages correspond to the ground truth defined through the labeling process, while the leftmost subimages are the detection attempt of the model. It is desired to use the ROS2 Object Analytics stack, which depends on the OpenVINO toolkit. The OpenVINO Model Optimizer provides a script<sup>3</sup> for converting TensorFlow inference graphs to IR. The Model Optimizer script produces the IR representation of the network, which can be loaded and inferred with the OpenVINO Inference Engine.

## 5.2. Robot Model and Simulation

### 5.2.1. URDF

The URDF of the KMR created in the specialization project is reused during this project. As new components are added to the system, these had to be added to the URDF as well. This includes the D435 cameras and the Robotiq 2F-85 gripper. The individual components can be seen in Figure 5.8.

The mesh and description of the D435 cameras are found at [69]. This project only includes a mesh for the visuals, hence, the collision geometry is described by a rectangular box. This component is reused four times in the URDF. One time at the manipulator, and three times at the mobile base. As the transformations received from the URDF are highly relevant for achieving correct sensor data, the new components must be attached in exact positions. Therefore, the adapter and plates produced, described in Section 5.1.1, are also included. They are modeled as a cylindrical plate and three rectangular boxes. The meshes and description of the Robotiq 2F-85 gripper are retrieved from [70]. The complete URDF of the KMR with all devices included can be seen in Figure 5.9.

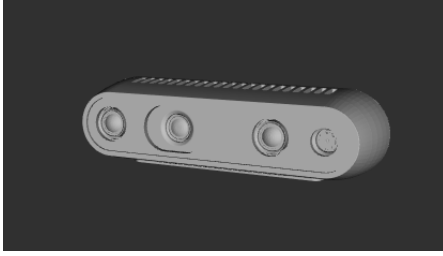
The Tf package keeps track of the frames in the robot system. Most of the transforms are retrieved from the URDF file. The frames related to the camera frames are published by the Realsense camera ode, while the transform between the odometry and the base of the robot, and the laser scanners and the scan frames are published by their respective data publisher.

### 5.2.2. Gazebo

Simulation has been an important part of this work, meaning a complete Gazebo model had to be created. The model is created with the same components as the

---

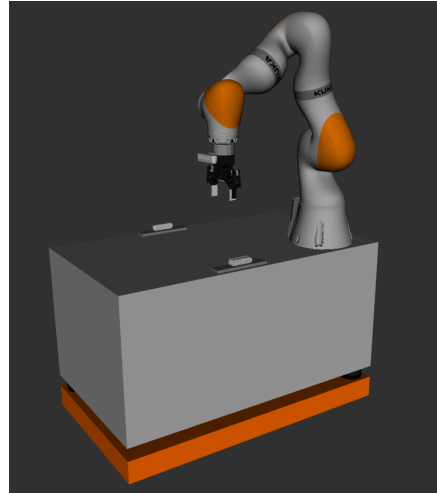
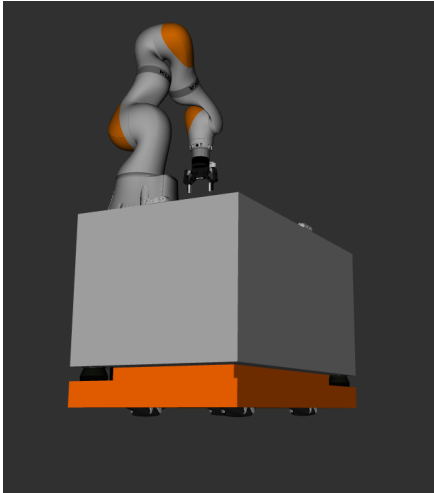
<sup>3</sup>[https://github.com/opencv/dldt/blob/2020/model-optimizer/mo\\_tf.py](https://github.com/opencv/dldt/blob/2020/model-optimizer/mo_tf.py)



(a) Model of the D435 camera.



(b) Model of the Robotiq 2F-85 gripper.

**Figure 5.8.:** Visualizations of the URDF models for the camera and the gripper.**Figure 5.9.:** The complete URDF model of the KMR with all devices included. This includes four Mecanum wheels, two SICK sensors, four D435 cameras, a Robotiq gripper and adapters.

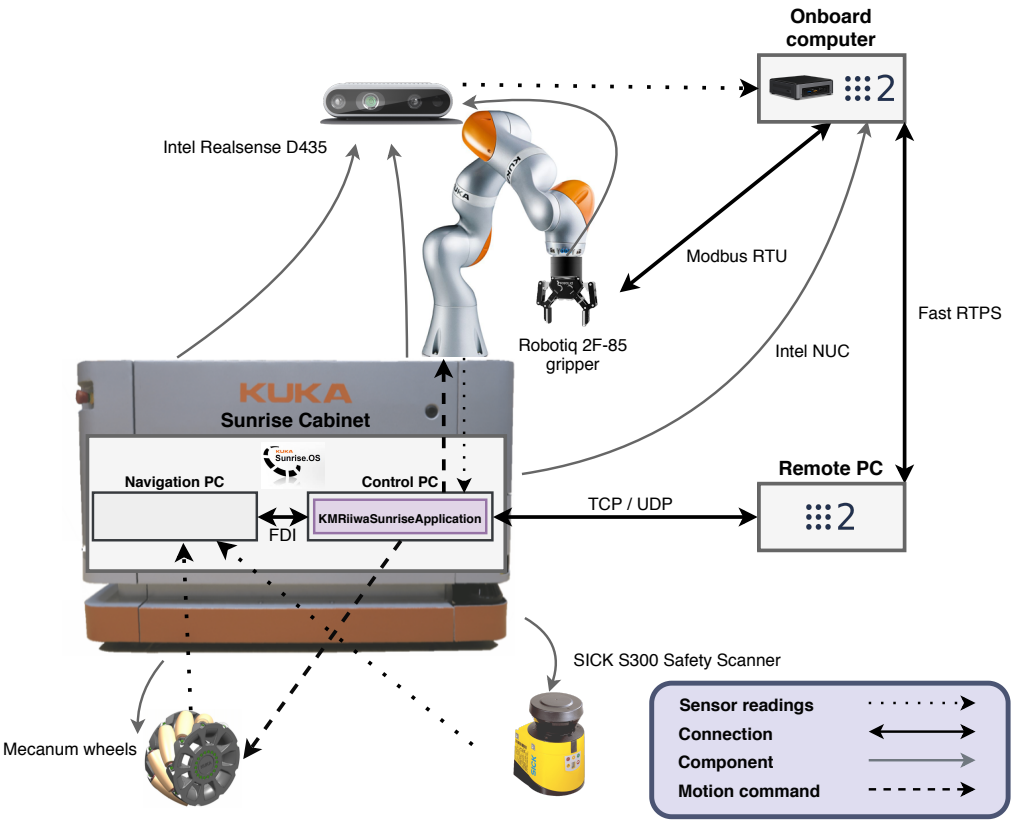
URDF except for the gripper, as the gripper was not relevant for the simulated experiments.

To have proper working simulated components, different plugins are added. The plugin `gazebo_ros_ray_sensor` is used for the SICK scanners and is configured to publish laser scan data to ROS. For the D435 cameras, the `libgazebo_ros_camera` plugin is used. In general, it publishes raw images and camera info for generic

camera sensors. The plugin was configured to publish raw depth images and point clouds. The `libgazebo_ros_joint_state_publisher` plugin publishes the state of joints in simulation. This is currently used for the four wheels, but do not have any vital functionality regarding the experiments performed in simulation. The plugin `libgazebo_ros_planar_move` is used to create the holonomic movement. It uses twist messages to move an entity in the XY plane and publishes odometry data, which makes it a good fit for experiments related to Navigation2.

### 5.3. Physical Architecture

The physical architecture of the robot system is illustrated in Figure 5.10. The KMR is the central unit, including the Mecanum wheels and the SICK S300 Safety Scanners. The other devices are installed on the robot to extend the functionality and abilities of the system.



**Figure 5.10.:** The physical architecture of the robot system with components and communication medium denoted.

The Sunrise application, `KMRiiwaSunriseApplication`, is installed on the control PC in the Sunrise Cabinet. It handles the communication with internal devices on the robot to execute motion and retrieve sensor data. Motions commands are sent to the joint motors of the LBR and the motors of the wheels on the KMP. Sensor readings from the Mecanum wheels and SICK lasers are retrieved at the control PC through a FDI connection to the Navigation PC, which is further connected to the components. The sensor readings in the joints are retrieved directly by the robot controller through internal couplings.

The Sunrise application running on the control PC further communicates with the remote PC through a TCP or UDP connection. The data from the four cameras are published to the ROS network by the onboard computer. The Robotiq 2F-85 gripper is mounted on the LBR and communicates with the onboard computer through the Modbus RTU serial protocol. The onboard computer communicates with the remote computer over the ROS network utilizing the Fast RTPS middleware.

## 5.4. Communication Architecture

A new communication architecture between the KMP and ROS has been implemented based on the drawbacks of the former architecture, mentioned in Section 1.2.1. According to the previously described definition of a ROS node, a node should have a specific purpose, and each node should handle one part. In the new architecture, the communication task is split into multiple subtasks, where each node is handling a separate subtask of the communication.

The separate tasks of the communication are:

- Sending commands from ROS to the KMP.
- Sending data from the SICK scanners from the KMR to ROS.
- Sending odometry data from the KMR to ROS.
- Sending status messages about the KMP from the KMR to ROS.
- Sending commands from ROS to the LBR.
- Sending status messages about the LBR from the KMR to ROS.
- Sending data from the sensors on the LBR to ROS. This data describes the position and torque of the seven joints in the manipulator.

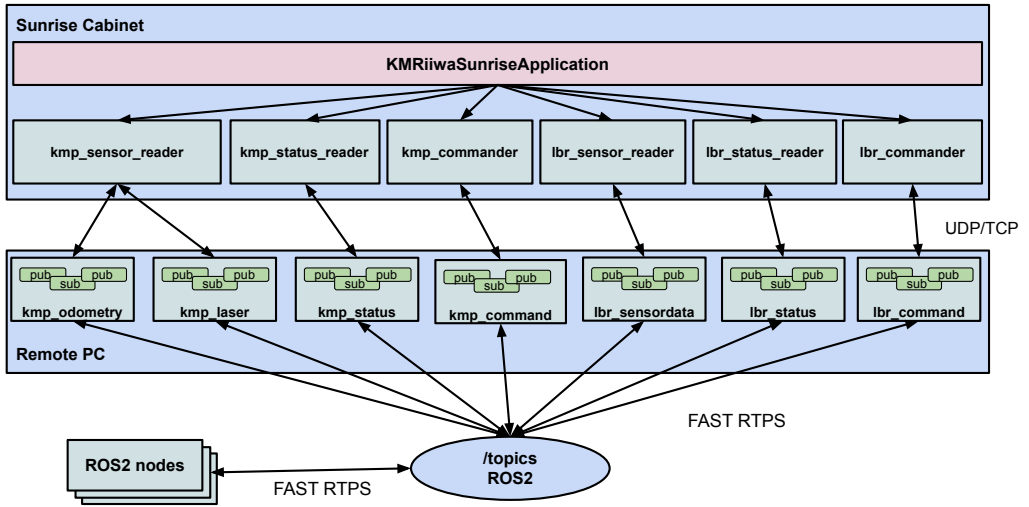
Each of these tasks has an associated node which handles the task, and are launched from the remote PC. Similarly, on the Sunrise Cabinet, separate Java programs are run to handle each task. For simplicity, these Java classes are also

referred to as nodes as they provide mainly the same functionality as the ROS nodes. Each of the ROS nodes creates a separate connection to its corresponding Java node. The connection can be either TCP or UDP, as both protocols are implemented. Messages between ROS and Sunrise are transmitted with the selected protocol as strings on the format shown in Equation (5.1).

$$\underbrace{Length}_{\text{Additional Information}} > \underbrace{Type \ (LaserID) \ Timestamp \ Data}_{\text{Message}} \quad (5.1)$$

The additional information, *Length*, is only included if the message is sent over TCP. As TCP is a buffer protocol, this is necessary to ensure that the whole message is read. The *Type* field describes what kind of data this message includes, and is necessary to know how the data should be processed. If the message is coming from the robot, an example of a type could be *Odometry*. An odometry message further contains a data field describing the pose and velocity of the robot.

The new architecture for communication between the KMR and ROS is illustrated in Figure 5.11. The solution is scalable, and more nodes can be added if necessary. This architecture is also described in the paper in Appendix B, which was written in March 2020.



**Figure 5.11.:** The architecture of the communication system between the KMR and ROS.

### 5.4.1. Remote PC

On the remote PC, there are seven different nodes available for communication with the KMR: `kmp_odometry`, `kmp_laser`, `kmp_command`, `kmp_status`, `lbr_sensordata`, `lbr_status` and `lbr_command`.

During the initialization of the nodes, they establish either a TCP or UDP connection to the Sunrise Cabinet. The connection type and the IP address of the robot are specified as parameters. The nodes either implement publishers or subscribers, based on the direction of the data. The nodes sending data from ROS to the KMR have subscribers, subscribing to the relevant ROS topic. The nodes receiving data from the KMR have publishers to publish the data to the correct ROS topic. All the publishers and subscribers and their associated ROS topics are listed in Table 5.1 and 5.2. Data read from the topics are formatted as ROS messages and need to be formatted as regular strings, while the data from the KMR are transferred as a string and need to be formatted as ROS messages before they can be published. Both types of nodes include data processing methods to format the strings the correct way.

The nodes initialize the `TCPSocket` and `UDPSocket` objects. The socket classes include functionality for connecting to the socket, closing the socket, and sending and receiving messages. A more specific description of the functionality for transmission over TCP and UDP is given in the specialization project report [3, p. 55-60].

The `kmp_odometry` and `kmp_laser` nodes handle the sensor information retrieved from the KMP. The `kmp_command` node subscribes to multiple ROS topics and forwards the commands from each topic. The currently supported commands are move with an absolute velocity, move to a given pose and shutdown. The shutdown command is essential to be able to shut down all connections and threads in the program correctly. The ability of moving with an absolute velocity is crucial for the use of Navigation2. Velocity commands and pose commands can be sent through an implemented keyboard, which can be run from the terminal by the user. The `kmp_status` node retrieves information from the KMP and stores the information in a ROS message named `KmpStatusdata`. This message type, which is shown in Table 5.3, is custom made for the interface and includes information that is useful when operating the robot. The `lbr_status` node has the same functionality, and publish the status data from the LBR as a custom `LbrStatusdata` message. The fields included in a `LbrStatusdata` message are shown in Table 5.4. The `lbr_sensordata` node handles sensor data coming from the LBR. The data includes the angle and measured torque of each of the seven joints of the manipulator. The information is combined into a ROS `JointState` message. This information is necessary to perform path planning for the manipulator.

**Table 5.1.:** Publishers for publishing data from the KMR to ROS

Name	Message type	Topic	Description
pub_odometry	Odometry	/odom	Odometry information.
pub_laserscan1	LaserScan	/scan_1	Data from B1 S300 laser (front).
pub_laserscan2	LaserScan	/scan_2	Data from B4 S300 laser (back).
pub_kmp_statusdata	KmpStatusdata	/kmp_statusdata	Information about the KMP.
pub_lbr_statusdata	LbrStatusdata	/lbr_statusdata	Information about the LBR.
pub_lbr_sensordata	JointState	/joint_states	Information about the torque and angular position of each of the seven joints in the manipulator.

```

# Goal #
trajectory_msgs/JointTrajectory path

---
# Result #
bool success
string error

---
# Feedback #

```

**Listing 5.1:** The MoveManipulator action for moving the manipulator along a planned trajectory.

The lbr\_command node handles commands to the LBR. Currently, it is possible to control the manipulator in two different manners. The first is to use one of the implemented keyboards which publish data the node subscribes to. From the keyboard, movement can be set for each joint, either to the left or right.



**Table 5.2.:** Subscribers for subscribing to data from ROS to the KMR

Name	Message type	Topic	Description
sub_twist	Twist	/cmd_vel	Make KMP move at a certain velocity.
sub_pose	Pose	/pose	Make KMP move to a certain pose.
sub_shutdown	String	/shutdown	Make the application on the Sunrise controller shutdown. Any string sent to this topic do the same purpose.
sub_manipulator_vel	String	/manipulator_vel	Make each joint of the manipulator move.

**Table 5.3.:** Fields included in a KmpStatusdata message

Name	Message type	Description
header	std_msgs/Header	Regular header for all ROS messages.
operation_mode	String	The KMR has three different operation modes. This field states the current mode.
ready_to_move	Boolean	True if the robot is ready to move, and no safety rules is violated.
warning_field_clear	Boolean	False if either of the warning fields of the S300 sensors are violated.
protective_field_clear	Boolean	False if either of the protective fields of the S300 sensors are violated.
is_kmp_moving	Boolean	True if the KMP is moving.
kmp_safetystop	Boolean	True if the KMP performs a safety stop. This happens if any of the internal safety monitoring functions of the Sunrise software are violated.

It is possible to move multiple joints at the same time, but not to control the velocity of the motion. The second way to control the manipulator is by providing

**Table 5.4.:** Fields included in a LbrStatusdata message

Name	Message type	Description
header	std_msgs/Header	Regular header for all ROS messages.
ready_to_move	Boolean	True if the manipulator is ready to move, and no safety rules is violated.
is_lbr_moving	Boolean	True if the LBR is moving.
lbr_safetystop	Boolean	True if the LBR performs a safety stop. This happens if any of the internal safety monitoring functions of the Sunrise software are violated.

it a trajectory. This method is used by the behavior tree, which is based on ROS actions. The node, therefore, implements a custom made ROS action called MoveManipulator to receive a generated trajectory. This action is shown in Listing 5.1. The goal is a trajectory for the joints, which are processed and sent to the LBR. The response is a boolean variable based on whether or not the request was a success. For the lbr\_command to verify whether or not the movement was a success, the node subscribes to the status data from the LBR published by the lbr\_status node. This status data includes the field is\_lbr\_moving. When this field is changed from true to false, it is assumed that the movement is completed successfully.

The lbr\_command node can also receive shutdown commands for the same purpose as for the kmp\_command node. Both nodes implement this functionality, as there may be cases where only the KMP or the LBR are in use.

### 5.4.2. Sunrise Cabinet

Six Java classes are implemented to communicate with the ROS nodes: kmp\_sensor\_data, kmp\_commander, kmp\_status\_data, lbr\_sensor\_data, lbr\_status\_data and lbr\_commander. The classes, hereby referred to as nodes, are responsible for sending sensor and status data, and for carrying out motion commands received from the ROS nodes.

Each node extends an abstract Node class containing conventional methods and variables, as the nodes have common functionality. The Node class is further an extension of the Java Thread class, making it possible to execute each of the communication nodes as threads.

Each node has an ISocket instance, which is an interface for socket objects. The

interface was created to ease the possibility of using different protocols. The two classes implementing the interface, `TCPSocket` and `UDPSocket`, contains functionality for establishing socket objects for the protocol options TCP and UDP, and for handling the socket objects and transmission of data. When the main application is launched, the nodes are initiated with the preferred protocol for communication and a port. Hence, each of the communication nodes establishes a socket object to transmit the data to the corresponding ROS node.

The `kmp_commander` receives commands from the corresponding ROS node. Two different types of motions can be executed. A pose command message includes coordinates of a pose relative to the robot. A predefined motion type from KUKA RoboticsAPI is used to move the KMP to the defined pose. Velocity motions are carried out by jogging the robot by utilizing methods from a `KMPjogger` instance. The `KMPjogger` class is implemented to handle the execution of motion and threads. For each new jogging execution, a new scheduled thread is started, and the class ensures they are terminated correctly to avoid the accumulation of threads. The `kmp_status_reader` uses functionality from KUKA RoboticsAPI to retrieve information from the robot. The status message is periodically sent to the remote PC, containing the information listed in Table 5.3. The `kmp_sensor_reader` class handles both the data from the SICK sensors and the odometry data, and establishes separate sockets for each. An FDI connection is established between the node and the Navigation PC, which makes it possible to subscribe to sensor data. A data listener class performs this subscription. The listener monitors the FDI connection and retrieves odometry data from the wheel encoders and scan data from the SICK scanners via the connection. When new sensor data are available, the data are transmitted to the corresponding ROS nodes for each of the data types. A method for automatic reconnection and subscription is implemented to handle cases where the FDI connection disconnects during program execution.

The `lbr_commander` receives messages containing motion commands from the corresponding ROS node, either with type `setLBRmotion` or `pathPointLBR`. The first type of motion message is shown in Equation (5.2) and can be used to move one or more joints of the LBR.

$$\underbrace{\textit{setLBRmotion}}_{\substack{\text{Message} \\ \text{Type}}} \quad \underbrace{\textit{JointIndex} \quad \textit{Direction}}_{\text{Motion specification for a joint}} \quad (5.2)$$

The `JointIndex` specifies which of the seven joints to be moved, and the `Direction` specifies if it is going to be moved to a negative angle (-1), positive angle (+1) or stop (0). When a joint is commanded to move, it continuously moves until a stop

signal is received or until the joint reaches its maximum angle. The other motion message type, `pathPointLBR`, is used for trajectories. A trajectory consists of points in space relative to the current joint configuration of the LBR. Each point in the trajectory is sent as an individual message from the ROS node, as shown in Equation (5.3).

$$\underbrace{\text{pathPointLBR}}_{\substack{\text{Message} \\ \text{Type}}} \quad \underbrace{\text{Type}}_{\substack{\text{Point} \\ \text{Type}}} \quad \underbrace{[\text{Poses}] \quad [\text{Velocities}] \quad [\text{Accelerations}]}_{\text{Joint specifications for a PTP motion}} \quad (5.3)$$

The point type defines whether the point is the start point, endpoint or a waypoint of the trajectory. Each of the joint specification arrays contains seven values, one for each joint of the LBR. The information received for a point is stored as a PTP-point instance, based on an object of the PTP motion type from KUKA Robotics API. If the point is of the type start point, the list containing the trajectory segments is cleared. Points of type waypoints are added to the list of segments. If the point is of the type endpoint, it is the last in the trajectory. A SplineJP motion is initialized with the segments representing the trajectory and executed asynchronously. A listener is added to the SplineJP instance, which updates the `lbr_status_reader` when the motion is finished.

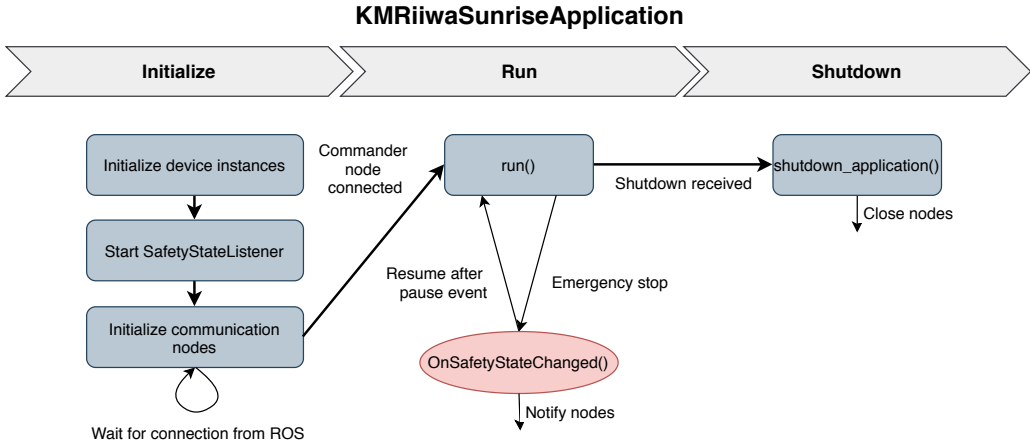
The `lbr_status_reader` retrieves information from the robot and transmits it periodically to the remote PC, similar to the `kmp_status_reader` node. The status message contains the information in Table 5.4. The `lbr_sensor_reader` periodically sends sensor data from the LBR to the corresponding ROS node. For each of the seven joints, the sensor data available is the joint position and the measured torque.

## 5.5. Sunrise Application

The `KMRiiwaSunriseApplication` is the application running on the Sunrise Cabinet, as shown in Figure 5.11. The `KMRiiwaSunriseApplication` is the overall component of the Java program, while the functionality is spread across the communication nodes. The main functions of the `KMRiiwaSunriseApplication` can be summarized with the following tasks, and are illustrated in Figure 5.12.

- Initialize the Java instances used to communicate with the actual devices.
- Initialize and handle the communication nodes.
- Handle behavior at emergency stops.

- Ensure that the program terminates properly.



**Figure 5.12.:** Flow diagram of the KMRiiwaSunriseApplication.

In the initialization phase, Java instances of the KMP, the LBR and the Sunrise Cabinet are initiated. As Java is an object-oriented programming language, it is important that the same instances are used throughout the whole program. Each of the communication nodes is further initialized with the necessary device instances, the desired type of protocol and the communication port to be used by the node. As mentioned, a node establishes a socket connection upon initialization and waits for a connection from the corresponding ROS node. The application will only continue running if one of the commander nodes have established a connection. This is done for safety reasons to ensure that the application can be terminated from ROS, as only the commander nodes can receive messages from ROS. If no connections are established to any of the commander nodes at the first attempt, the system will try to reconnect for a specified interval of time, and then terminate if the attempts are unsuccessful. If a connection to a commander node is successfully established, the application moves on to the run phase. The remaining nodes will only be started if their individual connection has been successfully established. This way, only the nodes with corresponding ROS nodes running are started, reducing unnecessary processes. If a Java node is not connected on the first attempt, it will try to reconnect periodically in the background. Therefore, if a ROS node is launched at a later time than the Sunrise application is started, the corresponding Java node will notice and connect to the node.

The KMRiiwaSunriseApplication initializes a `SafetyStateListener` class, which is implemented to notify the relevant nodes in the case of an emergency stop. If an emergency stop occurs, the status nodes transmit the information to the ROS node. The commander nodes implement a thread that continually listens to the

safety state and cancels the current motion, if any. When an emergency stop occurs, the application running at the Sunrise Cabinet is paused. The `KMRiiwa-SunriseApplication` is set as automatically resumable, which makes it possible to relaunch the application automatically. When an application is paused due to an emergency stop, the background application `AutomaticResumeBackgroundTask` will attempt to relaunch the application after a period of 3 seconds [30, p. 92]. If there are no violations of the safety restrictions of the system, the application will continue running from the place it was interrupted.

The `KMRiiwaSunriseApplication` is run until either `lbr_commander` or `kmp_commander` receives a shutdown message. Then, it moves on to the shutdown phase, and all of the nodes are notified. The communication nodes are closed by specific methods for each node defined in the individual classes.

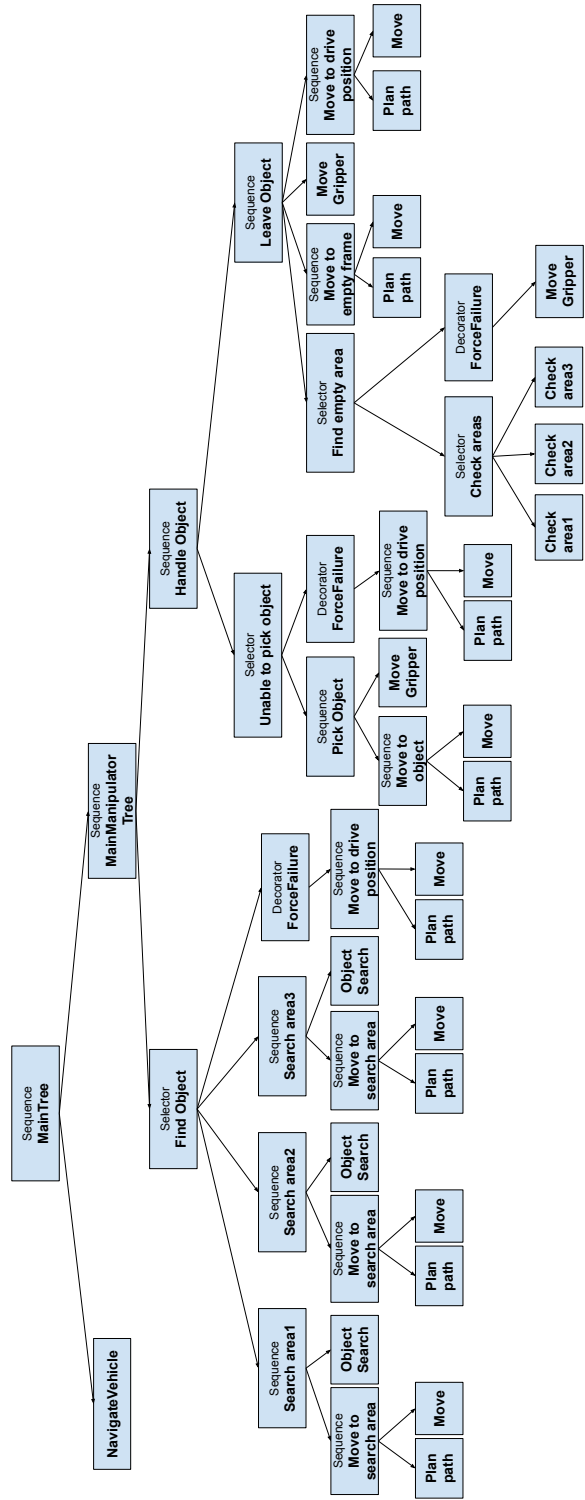
## 5.6. Behavior Tree

A behavior tree at the remote PC controls the logic carried out by the KMR. The logics include navigation of the mobile vehicle, movement of the manipulator, searching for objects by a camera and gripper functionality. The implementation is based on the `Behaviortree.CPP` library.

The leaf nodes of the tree are made as in a service-oriented architecture. Each leaf node acts as a client node that communicates with the associated server, which further performs the actual operation. An example of this is the `MoveGripper` node that communicates with the ROS gripper node, which further communicates with the actual gripper. As the behavior tree nodes are communicating with ROS nodes, the module named `nav2_behavior_tree` from `Navigation2` is reused for simplification. This module provides a C++ template class for integrating ROS actions into the behavior tree and also a generic `BehaviorTreeEngine` class that simplifies the integration of behavior tree processing into ROS nodes [71]. Hence, the behavior tree nodes and the ROS nodes communicate through ROS actions.

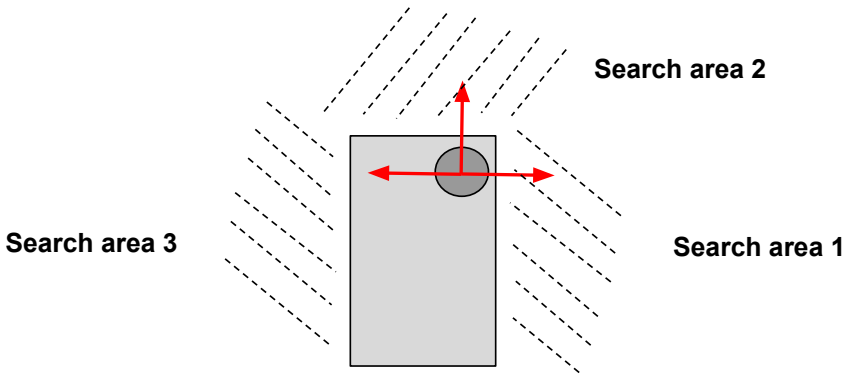
The behavior tree which will be presented in this section is created based on a fetch and carry scenario. It is illustrated in Figure 5.13, and is read from left to right. The tree is generic, and the nodes can be combined in different ways to get the desired behavior. This tree is an example of how the implemented nodes can be combined and carry out the following behavior.

The mobile vehicle navigates to a given goal pose. If the navigation is successful, behavior related to the manipulator is invoked. This behavior is split into finding an object and handling the object. The first task is to look for an object. There are three predefined search areas, illustrated in Figure 5.14. Search areas 1 and 3 are 90° to the left and right from the manipulator drive position, respectively,



**Figure 5.13.:** The implemented behavior tree. The composite and decorator nodes are denoted by names in the tree. Nodes without an explanation are primitives nodes, hence either condition or action nodes.

while search area 2 is  $180^\circ$  away from drive position. A path is planned to the given search area, the manipulator follows the path and starts searching for an object using the camera attached to the manipulator. If no object is found, the same logic is carried out for the next area. If an object is found, the subtree concerned with finding an object is successful, and the behavior for handling the object is invoked. If no object is found in any of the areas, the manipulator moves back to the drive position, and the tree fails. Handling the object consists of moving to the object and closing the gripper to pick up the object. If performed successfully, an empty place on the vehicle to put the object is looked up. If no empty place is found, the gripper opens again and leaves the object. If an empty place exists, a path is planned and executed to this place. The gripper opens to leave the object, and the manipulator is moved back to the drive position. If this is executed successfully, the whole tree is successful and completed.



**Figure 5.14.:** There are three predefined search areas for the manipulator. Search areas 1 and 3 are  $90^\circ$  to the left and right from the manipulator drive position, respectively, while search area 2 is  $180^\circ$  from drive position.

### 5.6.1. Tree Nodes

The behavior tree in Figure 5.13 consists of both primitive, decorator and composite nodes. A single decorator node is used, ForceFailure, which is a predefined node from BehaviorTree.CPP. This node will always return failure, independently of what the child node returns. In some situations, failure handling is necessary if the action fails. Such a situation is if no object is found in any of the search areas, the gripper fails to pick up the object or if no space was found to leave the object. If this happens, the tree should not fail immediately, but move the manipulator back to drive position. The failure handling might succeed, but the behavior tree should not be further processed as the original task failed. The ForceFailure node



makes sure the failure handling also returns failure, causing the whole tree to fail.

Two types of composite nodes are used, sequences and selectors. The BehaviorTree.CPP library provides different variants for both types of nodes. Among them is the Fallback node, which is a basic selector node and the one used in our implementation.

For the primitive nodes, one condition node and six different action nodes are implemented. The primitive nodes are custom made and preregistered, which means they can be reused as desired in different behavior trees. They communicate with the associated ROS nodes through custom made ROS actions. The custom made ROS actions are described together with the relevant ROS node in section 5.7. The implemented primitive nodes are PlanManipulatorPath, MoveManipulator, NavigateVehicle, EmptyFrame, MoveGripper and ObjectSearch.

PlanManipulatorPath is an action node that communicates with the RunMoveIt node. This node has two input ports, *plan\_to\_frame* and *object\_pose*. The first describes which frame the manipulator should be moved to based on a set of predefined frames. These frames are defined in the SRDF file. For situations where an object is to be grasped, the frame can not be predefined. A path to the object needs to be planned by MoveIt. Therefore, if the given *plan\_to\_frame* is *Object*, the input port *object\_pose* is used to define the pose MoveIt should plan to. The path returned is written to the output port *manipulator\_path*.

MoveManipulator is an action node which is connected to the LBR commands node. This node does always follow the PlanManipuatorPath in a sequence, which means this node reads the output path written by the planning node. The path is sent to the LBR command node through a ROS action. When this action returns success, the blackboard is updated with the current frame of the manipulator.

NavigateVehicle is an action node communicating with Navigation2. Navigation2 is handling the navigation and control of the mobile vehicle. The current goal pose is read from the blackboard and sent directly to Navigation2. Unlike the other behavior tree nodes, the navigation node uses a predefined action from Navigation2 named NavigateToPose.

EmptyFrame is a condition node that checks whether an area on the vehicle base is empty or not. There are three predefined areas at the base, named carryarea1, carryarea2 and carryarea3. The status of the areas are stored as variables in the blackboard, and when starting up, they are all set to true, meaning they are empty. When objects are picked up and placed in the areas on the vehicle, the associated variable in the blackboard is set to false, meaning they are not empty.

MoveGripper is an action node that communicates with the Gripper node and invokes the gripper to open or close. Which action to perform is set through the

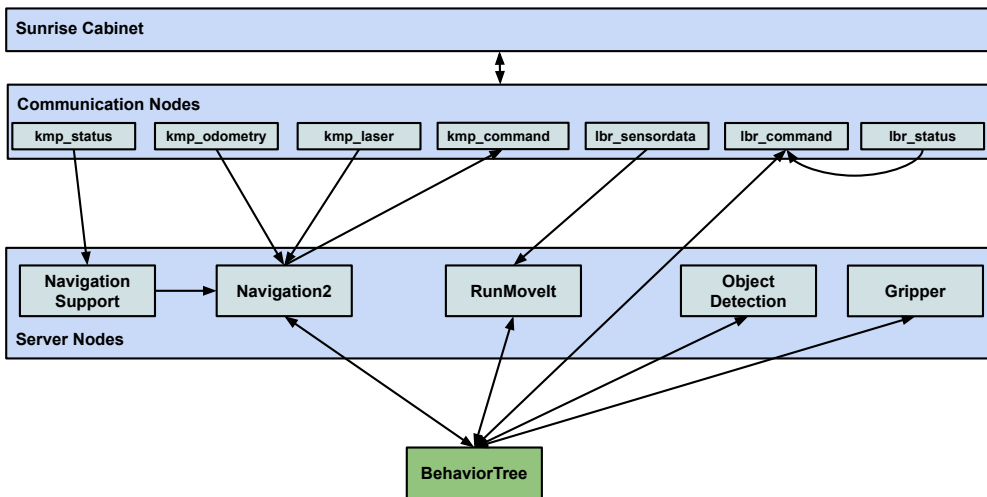
input port called *action*. If the gripper is opened and the current frame of the manipulator is either *carryarea1*, *carryarea2* or *carryarea3*, an object was left in this area. The blackboard is updated and the status of the area is set to false. The *EmptyFrame* condition node will then detect this frame as taken.

*ObjectSearch* is an action node that communicates with the *ObjectDetection* node. If the *ObjectDetection* node successfully finds an object, the pose of the object is written to the output port *object\_pose*.

## 5.7. Server Nodes

The leaf nodes in the behavior tree communicate with corresponding server nodes. These server nodes communicate with the hardware and external software.

Figure 5.15 gives an overview of the nodes and how they are connected to the rest of the system. The node executing the behavior tree is also described in this section, although it is not part of the server nodes. The *Navigation2* node is omitted from this section as it is an external node and not implemented by us.



**Figure 5.15.:** An illustration showing the server nodes and how they communicate with the rest of the system.

### 5.7.1. BehaviorTree Node

The *BehaviorTree* node is the superior controller of the system. It initializes the behavior tree by reading the XML description and passes it to the Behav-

iorTreeEngine class. The BehaviorTreeEngine class has two methods: `buildTreeFromText()` and `run()`. It uses BehaviorTree.CPP to create and execute the behavior tree.

When using navigation as a part of the behavior tree, the Navigation2 stack is dependent on receiving an initial pose for the robot. Usually, this is given by manually setting a pose in Rviz. For the solution to be fully autonomous, an initial pose is set during the initialization of the node. This is performed by a ROS publisher, publishing to the `/initialpose` topic, which is the same method Rviz uses to set the pose. The solution demands that the robot starts in the same position every time. When a robot is not in use, it is typically located at the charging station. Hence, this is considered to be a suitable solution. This position is defined in the BehaviorTree node's parameter file as the home position.

The BehaviorTree node has a method for initializing a new navigation goal pose. This is done by reading a list of goals from the parameter file and setting the first pose in the list as the current goal in the blackboard of the tree. In the parameter file, three work stations are predefined with coordinates, and the goal list consists of a combination of these work stations. The goal list can be changed, and additional work stations can be defined in the parameter file by the user. The behavior tree is executed by calling the `run()`-method from the BehaviorTreeEngine class. Independently of whether the execution of the tree fails or succeeds, another goal pose is initialized, and the behavior tree is executed again. The sequence continues until the goal list is empty. When the goal list is empty, the vehicle drives back to the home position.

### 5.7.2. NavigationSupport Node

The NavigationSupport node is independent of the behavior tree and is created as a support node for Navigation2. It supplements Navigation2 by dealing with the built-in safety restrictions of the KMR, which cause the problem described in Section 1.2.1.

Newly implemented behavior in Navigation2 makes it possible to update the parameters which specify the maximum velocities dynamically. The responsibility of the NavigationSupport node is to monitor the warning and protective fields of the KMP. This information is included in the KmpStatusdata message and is continuously updated. The node subscribes to status data and takes action if the parameter values are changed.

If the parameter defining if the warning field is clear changes from true to false, an object is detected close to the robot, and the speed must be lowered. If the same parameter changes from false to true, the robot has moved away from the

object, and the velocity can be increased. The velocity is changed by making a call to a ROS service within the Navigation2 controller server. This service is named `SetParameters` and is used to change parameters on the node at runtime. The relevant parameters are `max_vel_x`, `max_vel_y`, `max_vel_theta` and `max_speed_xy`. The parameters are changed between two static arrays denoting high and low velocity.

As the size of the warning and protective fields decreases when the velocity is reduced, the status of the warning field will immediately change back to true when the velocities are lowered. Similarly, when the velocities are increased, the size of the warning fields increases accordingly. This makes the warning field parameter, and hence, the velocities, jump back and forth between true and false, and high and low. A time threshold is used to avoid this behavior. The velocities are not changed if they have been changed within the last three seconds. After three seconds, the obstacle has most likely passed, or it is sufficiently close to remain in the warning field, such that the robot continues driving at low speed.

### 5.7.3. RunMoveIt Node

The `RunMoveIt` node handles motion planning for the manipulator. It takes advantage of the `MoveItCpp` interface, which configures and loads `MoveIt2` functionality. Further, the planning scene is initialized. As the KMR moves around in the MANULAB, the planning scene should be dynamically updated with sensor data. It was decided to neglect collision objects in the environment during the experiments to simplify the setup. The node has functionality for adding static collision objects, as it was widely used during testing and debugging.

`MoveItCpp` is also used to set up a planning component. A planning component class includes methods for setting goals and execute planning, based on the concepts described in Section 4.6.2. The planning component is defined to be *manipulator*, where *manipulator* is the chain going from the base of the LBR to an imaginary point in the middle of the gripper's fingers.

The node implements an action server named `PlanToFrame`, which the behavior tree node `PlanManipulatorPath` connects to. The `PlanToFrame` action is shown in Listing 5.2. The goal message consists of both a frame and a pose. The goal for the planning component is mainly set to be the frame, which defines a joint configuration specified in the SRDF file. The exception is when the frame is *Object*, where the goal is set to be the given pose. The imaginary point within the gripper is set as the link to end in this pose. For all other frames, the pose field is empty.

```

# Goal #
string frame
geometry_msgs/PoseStamped pose

---
# Result #
bool success
trajectory_msgs/JointTrajectory path
string error

---
# Feedback #

```

**Listing 5.2:** PlanToFrame action for planning the manipulator path to a given frame.

The SRDF file includes joint configurations for the frames driveposition, search1, search2, search3, carryarea1, carryarea2 and carryarea3. The search frames are the different configurations for performing object search, and the carry areas are different configurations for leaving an object on the vehicle base. It also defines the previously mentioned group *manipulator*.

When an action request is received, the goal is set, and the performed planning results in a trajectory. The response to the action request includes both a boolean value determining whether or not the planning was successful as well as the planned trajectory. The plan is visualized in Rviz if the program is run during execution.

#### 5.7.4. Gripper Node

The Gripper node contains functionality for controlling the Robotiq 2F-85 gripper using Modbus RTU. The messages described in Section 3.2.1 are used to communicate with the gripper. They are defined in the class GripperMsg, where each message is stored as an enum.

Initially, the gripper is activated, and further, the node has methods for opening and closing the gripper. The messages of these requests have function code 16 and data specified in Table 3.7. Common for both types of motion requests is that the bits **rGTO** and **rACT** are set to 1. The position byte, **rPR**, is either set to fully open or fully closed. When opening, full speed and full force is requested, while when closing, the force is set to half of the maximum value to account for fragile objects.

The Gripper node is connected to the MoveGripper behavior tree node through a ROS action. The action message is defined in Listing 5.3.

```
# Goal #
string action
---
# Result #
bool success
string error

---
# Feedback #
```

**Listing 5.3:** The Gripper action for opening and closing a gripper.

The goal message defines what action to perform, either open or close the gripper. The result message includes a boolean describing if the request was a success or not. This information is obtained by transmitting a message with function code 03 to the gripper and read the received response. The received response contains the data specified in Table 3.6. Further, the byte in the response corresponding to the gripper status is read. When opening the gripper, the motion is a success if the requested position is reached. In our project, the only purpose of closing the gripper is to pick up an object. With this in mind, successful closing of the gripper is defined to be when an object is detected between the gripper fingers. If the gripper fails to close around an object, the action request fails.

### 5.7.5. ObjectDetection Node

The ObjectDetection node contains functionality for controlling the object detection process.

The node depends on three other nodes, which must be launched at startup: a RealSense node, an OpenVINO node and an Object Analytics node. The RealSense node must be launched to publish image data from the D435 camera attached to the manipulator. The OpenVINO node starts an object detection pipeline. The object detection model to be used in the pipeline and the confidence threshold for the search are set as parameters. This node is responsible for recognizing objects in 2D. The Object Analytics node is launched to localize the detected objects in 3D.

As object detection is a computationally heavy process, it should be performed only when necessary. The pipeline manager in the ROS2 OpenVINO framework allows to control existing pipelines. This is done by sending calls to a ROS service in the pipeline node. The service can be used to run and pause the object detection pipeline. The Object Analytics node subscribes to the topic containing detected objects in 2D images. If no messages are published, the node will not search for 3D poses of objects.

As the previous nodes, this one is also connected to the behavior tree through a ROS action. The ROS action ObjectSearch is shown in Listing 5.4. When a request for an object search is retrieved, the object detection pipeline is started by a call to the pipeline service. The ObjectDetection node subscribes to the topic containing 3D bounding boxes of the detected object. If the probability for a detected object is above a specified threshold, the center point of the box is calculated and returned. A probability check is repeated in this step to ensure that the pose of the detected object is a good fit. The orientation for the pose of the object is set to a fixed value, which was found experimentally. If an object is found, the action request is successful. The outcome is returned in the action response as a boolean, together with the pose of the object.

```
# Goal #  
  
---  
# Result #  
bool success  
geometry_msgs/PoseStamped pose  
string error  
  
---  
# Feedback #
```

**Listing 5.4:** The ObjectSearch action to start searching for an object by the attached camera.





# Chapter 6.

## System Review

This chapter discusses and evaluates the implemented system. The evaluation focuses on the parts where important decisions or actions have been made during the development and parts which could have been improved. First, the communication architecture is considered in Section 6.1. Next, in Sections 6.2 and 6.3, a review of the communicating subsystems, namely Sunrise and ROS, is given. Section 6.4 discuss the external sensors and actuators applied in the work. Finally, remarks on the system architecture are given in Section 6.5.

### 6.1. Communication Architecture

The new architecture works in a satisfactory manner. The performed changes made it easy to add the new LBR nodes to expand the system. As was desired, nodes can be launched individually, which makes it straightforward to test separate parts of the system. For the Sunrise implementation, the ISocket interface and the abstract Node class makes it simple to use different socket types or add more communication nodes. This confirms that the architecture makes the system more scalable and flexible.

In terms of a more fault tolerant system, the individual nodes make it possible for the system to keep running if one node stops working. This means, for example, that you can still send shutdown commands and close the system properly, even though one of the sensor nodes has disconnected due to an error. What should have been improved is error handling and displaying more messages to the user. A common problem during experiments has been that one or multiple nodes stop sending data because of latency in the network. Currently, this can only be discovered by monitoring the topics where the data is published. A check could be implemented to notify the user if more than a certain time has passed since a message was received.

A main problem with ROS is how to maintain single point-of-control. Any open interfaces that can lead to undesired robot motion initiated by an attacker will be considered a safety issue. This is both a risk if someone connects to the communication ports of the robot, and also if an attacker publishes messages to the ROS motion topics. The first problem could be solved by configuring the Java nodes to accept connections only from a registered or identified node. As the concept of ROS topics is anonymous publishers and subscribers, the second problem is more complicated. One solution is to validate the data before it is sent to the robot. As for motion commands, there could be a check to verify if the velocity is within specified limits. Such validations could prevent extreme values, but would lead to a less generic system. This is a disadvantage if you want to use the same communication architecture for multiple robots.

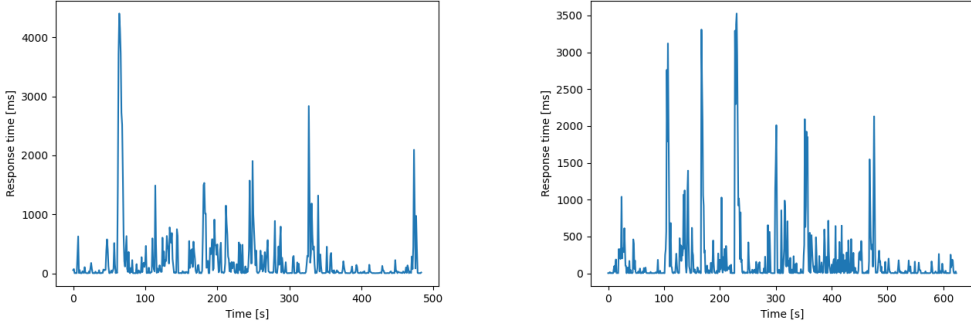
### 6.1.1. Communication Protocol

The communication between ROS and Sunrise is based on [TCP](#). As stated in [Chapter 5](#), options for both the [UDP](#) and TCP protocol are implemented. The Sunrise system only has ten ports for communication with external devices [[25](#), p. 410]. Through experiments, it has emerged that incorrect termination of the program causes the UDP ports to be unavailable for several hours. As the nodes of the implemented system communicate over seven different ports, the system depends on the ports being available at all times. Therefore, TCP has been used, even though UDP is the preferred communication protocol for real-time applications. TCP transport includes unnecessary overheads and requires acknowledgments from the receiver. Compared, UDP is more efficient in terms of bandwidth and less demanding of poor-performing networks. UDP does not ensure that the message arrives at the receiver, but message loss is not critical in applications where data is sent frequently. On the other hand, UDP entails a risk for shutdown and motion commands not to arrive, which should be taken into account. Switching to the UDP protocol should be done when the system and network are stable and there is no risk of unexpected errors.

### 6.1.2. Network

Throughout the experiments, instability in the network was observed, which influenced the collection of results. Both the external computers and the KMR must be connected to the same network for transmission of data. The network is affected when heavy programs are run on the external computers. [Figure 6.1](#) shows the response time over the network captured during two time intervals of around 10 minutes. The experiment conducted in both cases includes launching programs related to communication with the KMR, navigation, perception, object

localization and manipulation. The network traffic was captured by sending continuous pings to the robot controller during the experiments. When no programs are launched on any of the computers, the average response time is 3 ms. In the captured data traffic, the average response time is around 200 ms. A package loss of around 50 % was observed during the execution.



**Figure 6.1.:** The response time in the network captured during two experiments that included launching of heavy programs.

The consequence of overloading the network depends on the communication protocol. For TCP communication, the transmission of data over specific ports stops when no acknowledgment is received from the listener. For UDP communication, data packets are lost when delays occur. However, stagnation in the transmission of data is highly undesirable in a system where it is critical that commands are received and where sensor data should arrive in real-time. If the problem persists, further research should be conducted.

## 6.2. Sunrise

### 6.2.1. KMP Sensor Data

The sensor data of the KMP is transmitted from the Navigation PC to the Control PC through a FDI connection. The FDI connection class is included in a JAR file from the KUKA Navigation Solution software. If there is a latency larger than 3 seconds in the system, the connection automatically disconnects, which stops the transmission of sensor data. In addition to complex programs running on the network, the large number of threads running on the Control PC may be a source to latency. To overcome this problem, an automatic reconnection was implemented. This worked to a certain extent, but in situations where the FDI connection repeatedly disconnected, the sensor data would not be transmitted.

The FDI connection class has a variable, `DATA_TIMEOUT`, which controls the timeout leading to disconnection. The field is declared as *private final* and, hence, can not be changed from an external class. Although it has been preferable not to change the Sunrise system, the only solution was to do a minor change in the source code of the FDI connection class. The variable `DATA_TIMEOUT` was declared as *public* to enable accessibility from external classes. Further, the value of the variable was increased to 12 seconds to avoid disconnections. The adjustments were successful, and disconnections were only observed in scenarios where the delays in the network were too significant to operate the KMR in general. However, in these cases, the FDI is automatically reconnected due to the implemented solution.

### 6.2.2. KMP Motions

When executing all the Java nodes, nine threads are running, in addition to the ones established for the jogging motion. A single core on the Control PC in the Sunrise Cabinet is assigned to the application server. In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution. Concurrency indicates that more than one thread is making progress, but the threads are not actually running simultaneously. The switching between threads happens quickly enough that the threads appear to run simultaneously [72]. The jogging thread handled by the `KMPjogger` executes the motion commands periodically at a fixed rate. With fewer threads running, the default rate of 50 ms was sufficient for smooth motion. As the system was expanded with several more processes, it was observed that the motion of the KMP became jerky. This may have been caused by the increased number of threads running. The velocity motion became smoother when the rate of the jogging thread was lowered to 1 ms, but the robot still suffers from insignificant chopping. However, the Java implementation has potential for improvement. In general, the program suffers from unnecessary processes which may cause delays in the system. Threads are created for monitoring critical variables and connections and to notify other parts of the system. These could be replaced by Java concepts such as events and listeners to reduce processing. The `RoboticsAPI` provides useful functionality for this purpose, such as event triggers [26]. Per today, the sensor and status data from the LBR is sent occasionally from a thread based on a time threshold. Background applications can be utilized for reading and transmitting sensor and status data with a fixed delay. Another solution could have been to only transmit the data if any values are changed. However, if the communication protocol is changed to UDP, this might be a bad solution, as it is not ensured that ROS receives the updates.

### 6.2.3. LBR Motions

As described in Section 3.1.4, there are several options for executing motions for the manipulator. Two different types of motions are implemented, one for moving individual joints and one for following trajectories. The first motion type has been useful for testing the program and for manually moving the arm when the system is executed. The second type of motion is required to execute the planned trajectory from MoveIt. A trajectory consists of several points, which can be programmed as a group of individual motions.

Initially, it was tested to add each of the points of the trajectory to a list and execute each segment as a PTP motion. This led to jerky motions, as the distinct motions were not linked together. An option considered was to group the PTP motions in a Motion Batch. This type of grouped motion uses an approach for approximate positioning that leads to deviations from the plan. Therefore, a JP spline block was a more suitable option. Examples in the documentation of the Sunrise system shows how to make a new spline motion by defining each point of the trajectory when the instance is created [25, p. 372]. This approach is not applicable in our case, as an unknown number of points are received from ROS. The implemented solution is based on the approach used in the work of Virga and Esposito [6], using a list of PTP segments. The method makes it possible to create trajectories of different lengths that lead to continuous motions. However, it does not make it possible to execute the plan from MoveIt in real-time, as the trajectory is not executed until the last point is received.

If the LBR is commanded to move to its current joint configuration, hence, it receives a trajectory with no points and velocities, no motion is carried out. In the `lbr_commander` node, a listener is implemented, which notifies the `lbr_status` node when a trajectory is finished. When no motion is executed, the listener is not invoked, and the ROS nodes are never notified that the execution of the trajectory is finished. A simple check could have been performed to compare the received endpoint configuration to the previous or current joint configuration. If they are similar, the variable denoting if the execution of a path is finished should be set to true, such that the logic on the external computer is aware that the robot is in the commanded position.

### 6.2.4. Safety

An important focus during the work was to make the system universal. It should be easy for a user not familiar with the Sunrise software to use the system to control the robot, and the system should be applicable to any KMR. At the same time, it was desired that the ROS integration should interact with Sunrise as it is, without making any changes to the system. Even though, two of the rows in

the Customer PSM table have been changed, as the adjustments were considered as necessary.

## Manipulator Work Space

The default workspace of the manipulator is defined above the mobile platform, such that the LBR is not able to collide with the base. In horizontal space, the manipulator is not allowed to move outside the projection of the base, as this is an unknown environment. Increasing the workspace entails a risk, and collision avoidance should be handled. It still had to be done for the manipulator to be able to pick up objects from work stations. Initially, the lower boundary of the workspace was set to  $z=0$ , corresponding to the surface of the KMP. The boundary was later adjusted to a negative z-value, such that the manipulator could grasp objects from levels lower than the KMP. Setting the boundary of the workspace lower than the base is a risk, as the arm could collide with the base. This entails that the path planned by MoveIt is fully trusted.

## Protective Field

One of the challenges addressed in Section 1.2.1 was how to interact Navigation2 with the built-in safety functionality of the Sunrise system. More specific, the safety function that triggers an emergency stop if the protective field is violated. For safety reasons, the velocities have been kept low during the experiments with the KMP. As listed in Table 3.1 and 3.2, the sizes of the warning and protective field are approximately similar for low velocities. As a consequence, there is no time to reduce the velocity of the robot when an object enters the warning field before the emergency stop is triggered. The planned approach to solve the problem was to expand the fieldset of the SICK scanners. The fields can be configured with the SICK Configuration and Diagnostic Software, CDS. By adding an outer warning field, one could be notified of an object approaching, and the velocity of the robot could be reduced before the object entered the original warning field. Three steps are required to configure the fieldsets. First, the new monitored areas must be defined using the software SICK CDS. The outputs from the scanner must be configured as inputs to the Programmable Logic Controller, PLC, of the robot. This must be done with the TIA Portal and STEP 7 Advanced software provided by Siemens AG. The software can be used to expand the devices and safety functions of the bus system. The outputs from the PLC can further be mapped from the PROFINET bus using KUKA WorkVisual. The last step is required to access the I/O values in Sunrise Workbench. The advanced edition of TIA Portal requires a license. A request for this was sent in March, but the license was never obtained.

The described solution was a main focus of the work until the middle of the semester. During the spring, the MANULAB was closed over a longer period, which prevented the original solution to be carried out. Instead, an alternative solution had to be found that could be researched and implemented outside the laboratory. The Radio Control Unit, mentioned in Section 3.1, has a button for muting the protection fields. The functionality could have been useful for turning off the monitored fields when necessary. This could, for instance, be to drive away from an obstacle after an emergency stop was triggered or to approach work stations and narrow passages. However, it was found that I/O mappings, according to the previous description, were necessary to be able to implement muting from a Java program. The final solution was to deactivate the row Protection Field Status in the Customer PSM table. This row defines the safety functions related to a violation of the protective field. When disabled, the safety fields are still monitored, but do not trigger an emergency stop in the controller. The Java program of the Radio Control Unit uses a scaling factor to adapt the maximum velocity according to the status of the safety fields. This is similar to the implemented approach, where the maximum velocities of the Navigation2 controller are dynamically adjusted conforming to the status of the monitored fields.

As for the solution for the LBR, deactivating the safety functionality of Sunrise entails relying entirely on the obstacle avoidance functionality of Navigation2. Based on the results achieved with Navigation2, we concluded it was safe a safe solution.

Safety is one of the cores of the KUKA systems, and it has been a complex process to work around all of the restrictions. Endless time has been spent to understand the system and searching for solutions to work with the built-in safety functionality of the KMR. As time and resources were not enough for the solution with extended warning fields, a simpler solution had to be chosen.

## 6.3. ROS

### 6.3.1. Actions

For the actions utilized to communicate between the behavior tree nodes and the server nodes, the validations of success and failure could be improved, and the feedback functionality is used to a small extent. As for moving the manipulator, the action is presumed to be successful when the status field is `_lbr_moving` changes from true to false. This is not a proper validation as it does not ensure if the arm has followed the trajectory and reached the desired pose. Feedback could have been used to validate the joint states and make sure the manipulator end in the requested position. This also applies to when the gripper is closing. As of now,

the action is a success if the gripper is closing around an object. This is verified by checking if the force sensor detects any resistance between the fingers when the movement is finished. In scenarios where the orientation of the gripper is slightly incorrect, one of the gripper fingers tends to be bent over the object. When the gripper tries to embrace the object, a force is detected from the finger that is not bent and the closing motion stops. This leads to incorrect object detection, which is not handled by the behavior tree. Ideally, the check should be continued when the manipulator starts to move, to make sure the object was properly gripped. The parallel nodes in the behavior tree could be utilized to perform two actions at the same time. Another solution could be to define a more significant force for the closing motion of the gripper. This could force the gripper to continue closing, such that it becomes aware that it has missed the object. This solution depends on the fragility of the object to grip, but could be a proper approach in our case as the object is solid.

### 6.3.2. Programming Language

Most of the code running on the external computers are written in Python. The BehaviorTree node and the RunMoveIt node are written in C++, as to the APIs are written in this language. A system written in multiple languages might be difficult for others to use if they are not familiar with both languages. The system was initially created in Python, as this is the language the authors are more familiar with. A possible idea is to rewrite the system into C++. Further implementations could then be simplified, as the ROS client library for C++, `roscpp`, have a wider range of implemented functionality, and most example code for ROS2 is written in C++. In general, C++ is commonly used for performance and hardware-centric control, while Python is used for productivity [51].

## 6.4. Sensors and Actuators

### 6.4.1. Robotiq 25-85 Gripper

The Robotiq 25-85 gripper is part of the available equipment in stock at the MANULAB. The coupling is not optimal and has disadvantages associated. For example, it can lead to windings around the manipulator or the cable can get hooked up.

The Robotiq Universal Controller can be configured as an EtherCAT device. As the media flange of the LBR has an EtherCAT interface, it would be a more elegant solution to reconfigure the controller to use EtherCAT instead of Modbus RTU. This solution would require a cable for connecting to the EtherCAT port on the media flange. More specific this is a M8 connection with four poles, which was



not available at the laboratory. There is also an EtherCAT port located at the base of the robot arm that could have been used to connect the Robotiq controller. These configurations would still require cables routed along the manipulator to the Universal Controller, as the gripper itself is not compatible with EtherCAT, and the controller can not be mounted at the flange.

According to [73], the Robotiq 3F Adaptive Gripper is a good match with the LBR, as it is compatible with EtherCAT. The best solution would be to apply the 3F gripper by connecting it directly to the media flange. With this option, it would be possible to pass the cabling through the inside of the LBR. When connecting the gripper through the EtherCAT interface, the I/O signals can be configured in KUKA WorkVisual. The configuration requires few steps and is described briefly in [74]. This would overall lead to a more presentable solution, where the gripper could be controlled from Sunrise instead of ROS. In addition, it would be included in the safety restrictions of the robot, such that collisions between the base and the gripper is prevented. In terms of architecture and control logic, this solution would make more sense, as the gripper is an extension of the KUKA robot system. The KUKA RoboticsAPI contains methods for controlling different tools and would require less code than the current solution. The current implementation is less generic as it is specific for the 2F-85 gripper with Modbus-RTU, and can not be used with other types of tools.

#### 6.4.2. Intel Realsense Cameras

The ROS2 wrapper for the Realsense cameras contains bugs that were revealed during the work. Each of the camera nodes publishes images in the optical frames of the camera. Further, the nodes publish transformations from the base frame to the optical frames. The name of the base frame of the camera can be specified as a parameter to the node, but there is no such functionality for the optical frames. With three cameras installed on different poses on the robot, this resulted in images taken from different points of view being published in the same camera frame.

Another bug was discovered when two new cameras were purchased during the spring. When a camera node is launched, the serial number must be specified in the launch description. The Realsense node saves the serial number as an integer, and as our new cameras have serial numbers starting with '0', this is removed when the number is read. The node is not able to connect to the cameras as the given serial numbers are incorrect.

The bugs were solved by creating a fork of the original repository and changing the code. Additional parameters were added to the node to specify the names for the optical frames. For the serial numbers, the Realsense node and launch file are

changed to read and provide the serial number as a string instead of an integer. That way, the '0' at the beginning of the serial number remains.

## 6.5. Remarks

A focus for the whole system has been to make it scalable and flexible. The communication architecture can easily be extended with more nodes, both on the ROS and Java side. Within the behavior tree, other behavior nodes can easily be added as plugins. The already existing nodes are flexible by utilizing ports to get input and provide output. Different behavior can be achieved by creating different combinations of the behavior tree nodes. The external sensors and actuators can be replaced with different types, and a new ROS node with device-specific implementation must simply implement the ROS action that communicates with the behavior tree. The object detection model can be replaced by simply changing a parameter.

Overall, the architecture of the entire system is designed so that it can easily be adapted to specific needs.

# Chapter 7.

## SLAM

The ROS2 stacks Cartographer and RTAB-Map have been applied to create 2D occupancy grid maps by the SLAM approach. Both stacks have options for lidar and visual odometry, and for different sensor configurations for creating maps. It is desirable to exploit as much of the available sensor data as possible to fully capture the environment. The communication nodes related to the KMP are utilized for retrieving sensor data and moving the robot in the environment. The experimental environment is described in Section 7.1. Section 7.2 presents the results obtained, related to estimation of odometry and creation of maps. The results are evaluated in Section 7.3 and concludes which stack and sensor configurations that provides better results.

### 7.1. Experimental Environment

The experiments are performed at the MANULAB at NTNU in Trondheim. The area can be seen in Figure 7.1.

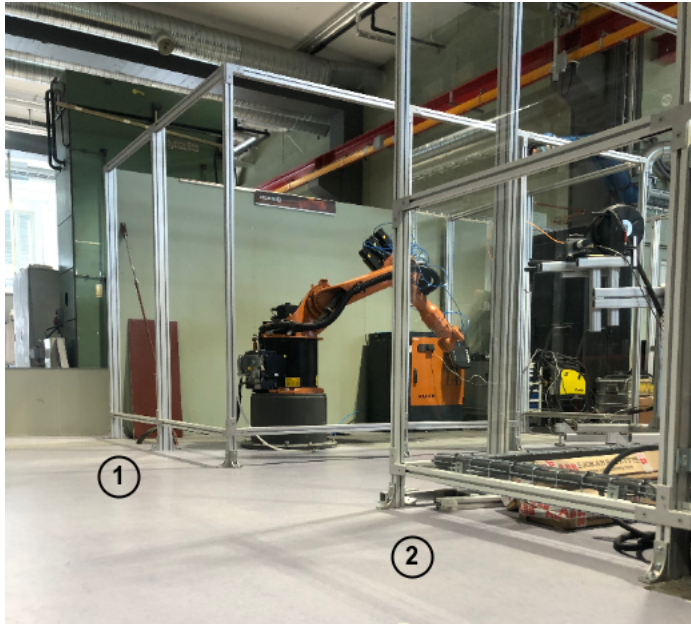
#### Robot cells

The robot cells were used for validating the performance of SLAM during the specialization project, and is used to validate the improvements in the results. The robot cells are shown in Figure 7.2.

The MANULAB is characterized by separated areas containing many objects and obstacles, constituting to a complex environment that is complex to map accurately. On the other hand, this contributes to a feature-rich environment suitable for VSLAM.



**Figure 7.1.:** The MANULAB at NTNU used for testing of the system.



**Figure 7.2.:** Robot cells in the MANULAB used for testing during the autumn.

## 7.2. Results

The maps provided in this section are created based on a single ROS bag of sensor data to be able to compare the maps correctly. The blue lines in the maps mark the trajectory of the KMR.

### 7.2.1. Odometry

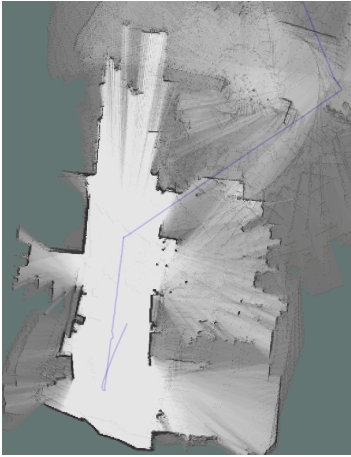
Lidar odometry and visual odometry are evaluated by not providing odometry from the wheel encoders. The KMR followed an approximately straight path. The trajectory is shown in Figure 7.3, and is based on the odometry from the wheel encoders. This is referred to as the ground truth when evaluating the odometry from range sensors.



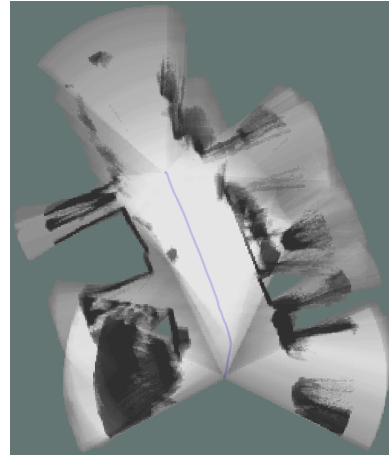
**Figure 7.3.:** The trajectory followed by the KMR down the hallway, based on wheel encoders, referred to as the ground truth. The map is created by RTAB-Map.

### Cartographer

As Cartographer uses point clouds and laser scans as input, visual odometry is not the correct theoretical term. Even though, as the point clouds are data from the cameras, we have chosen to separate the odometry into lidar and visual. Figure 7.4 shows the separated results for lidar odometry and visual odometry. Figure 7.5 shows the estimated odometry using both lidar and point cloud data as input. When using multiple input sources to estimate odometry, they are all given equal weight.

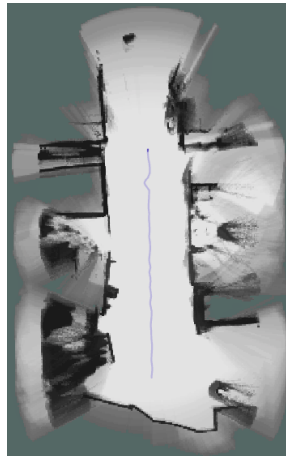


(a) Sensor input: two scans.



(b) Sensor input: three point clouds.

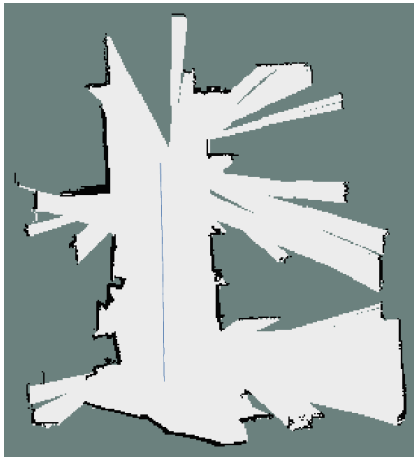
**Figure 7.4.:** Estimated lidar and visual odometry.



**Figure 7.5.:** Estimated odometry based on scans and point clouds.

## RTAB-Map

RTAB-Map has multiple nodes for computing odometry based on sensor data. Each node uses one type of sensor data, and only one odometry node can be used at a time. Hence, odometry based on lidar data and camera data is calculated in two separate experiments, shown in Figure 7.6.



(a) Sensor input: two scans.



(b) Sensor input: three RGB-D cameras.

**Figure 7.6.:** Estimated lidar and visual odometry.

### 7.2.2. Mapping

The mapping processes have been conducted with different sensor inputs, depending on the sensor configurations available as input. Two results are provided for each configuration, based on what was considered interesting for comparison. Some of the configurations are shown with and without loop closure, while different parameter settings are shown for others.

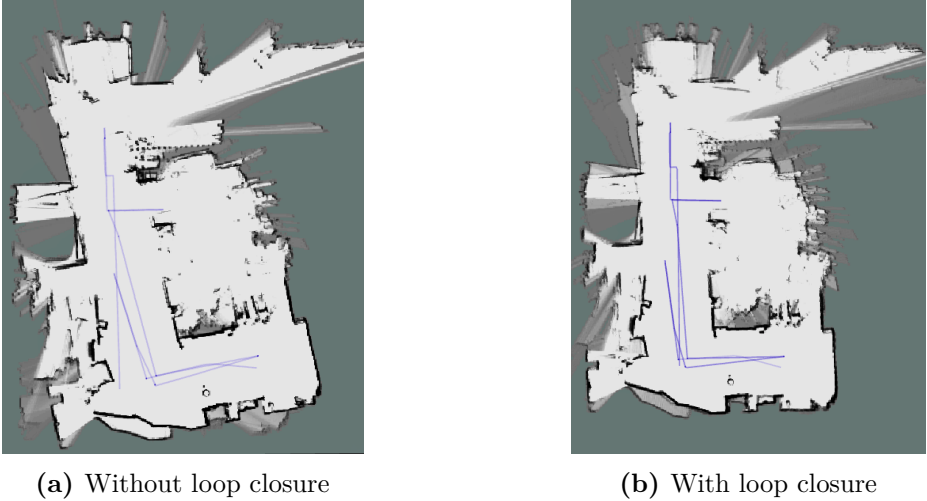
#### Cartographer

The four following sensor configurations were tested when creating maps with Cartographer:

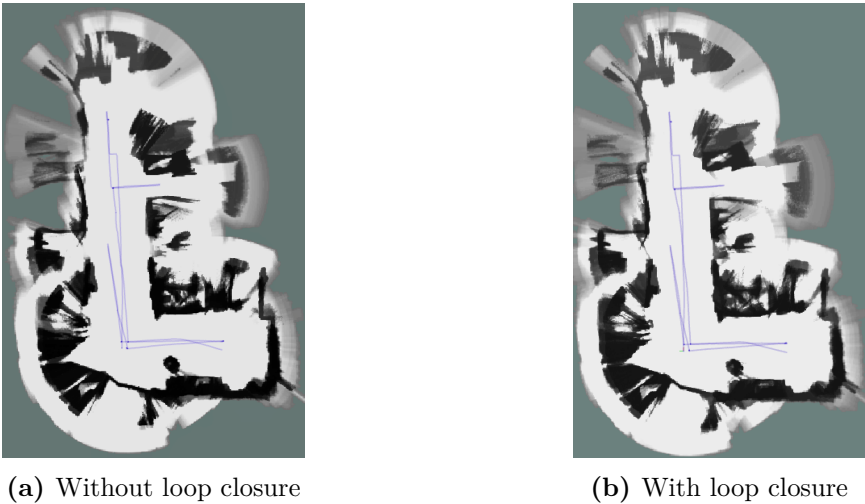
1. Two 2D scans.
2. Three 3D point clouds.
3. Two 2D scans and three 3D point clouds.
4. Five 2D scans from lasers and cameras, where the images from the cameras are converted to scans.

Projecting the depth images from the D435 cameras to 2D scans are obtained by utilizing the ROS package `depthimage_to_laserscan` [75]. In addition to the range measurements listed, odometry from the wheel encoders was used in all configurations.

The maps obtained with the different sensor configurations are shown in Figures 7.7 to 7.10.



**Figure 7.7.:** Maps created with two scans as input. The maximum depth of the measurements are set to 15 m.

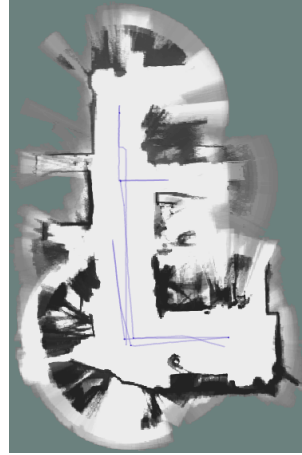


**Figure 7.8.:** Maps created with three point clouds as input. The maximum depth of the measurements are set to 4 m.



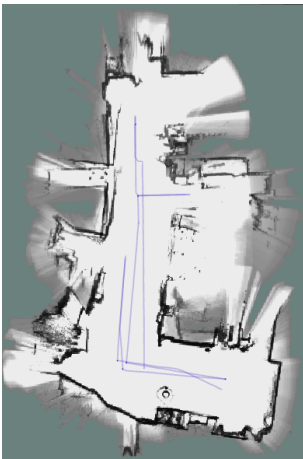


(a) Without loop closure.



(b) Without loop closure and increased size of submaps.

**Figure 7.9.:** Maps created with two scans and three point clouds as input. The maximum depth of the measurements are set to 4 m.



(a) Without loop closure.



(b) With loop closure and increased size of submaps

**Figure 7.10.:** Maps created with five scans as input, where the depth images have been converted to scans. The maximum depth of the measurements are set to 4 m.

## RTAB-Map

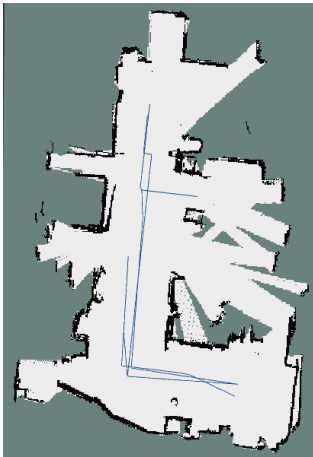
When creating a 2D occupancy grid map with RTAB-Map, the map is created by using either lidar data or camera data, but not both at the same time. If using both lidar and camera data, the map will be created from the scan data, and the camera will only be used for loop closure detection. When using laser scans as input, it is only possible to subscribe to one topic at a time. To utilize both of the scanners, the lidar data can be fused into a single point cloud, and be given as a laser scan point cloud input. This is done by utilizing the ROS package `pointcloud_to_laserscan` [76], which also includes a node for the inverse transformation.

If both laser data and camera data is present, the data utilized in the map can be controlled by the parameter *Grid/FromDepth*. By setting it to true, the map will be created from only the camera data. RTAB-Map provides a multi-session mapping functionality of which makes it possible to continue on a previously saved map in another session. That way, the parameter *Grid/FromDepth* could be changed between the sessions.

The following sensor configurations were tested:

1. Two 2D scans fused into one point cloud.
2. Three RGB-D images.
3. Three RGB-D images and two 2D scans, used in separate sessions (multi-session mapping).
4. Two RGB-D images and two scans fused into a single point cloud.

The maps obtained with the different sensor configurations are shown in Figures 7.11 to 7.14.

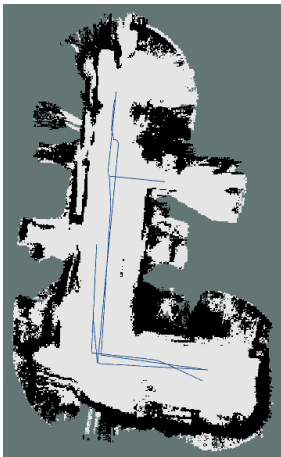


(a) Without loop closure

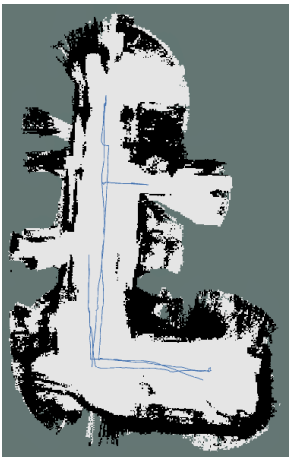


(b) With loop closure

**Figure 7.11.:** Maps created with data from two lidars. The maximum depth of the measurements are set to 15 m.

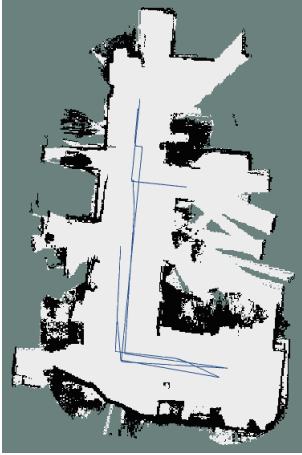


(a) Without loop closure

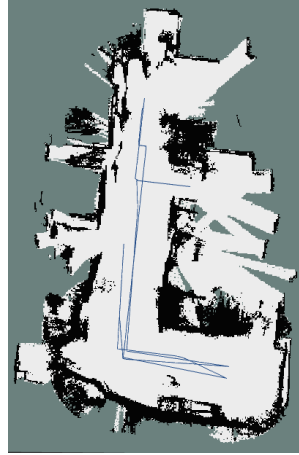


(b) With loop closure

**Figure 7.12.:** Maps created with image data from three cameras. The maximum depth of the measurements are set to 4 m.

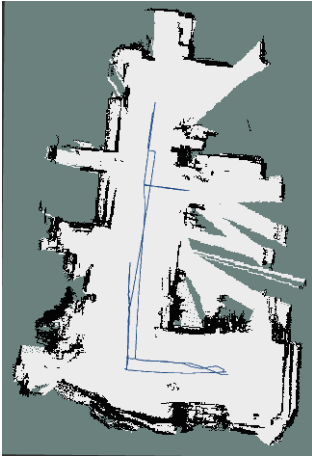


(a) First session: cameras. Second session: scans

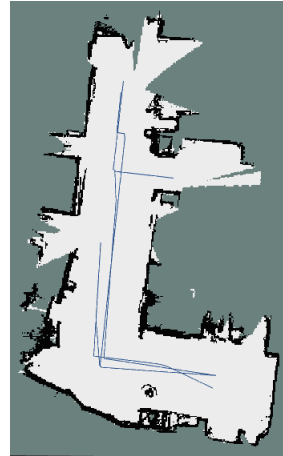


(b) First session: scans. Second session: cameras

**Figure 7.13.:** Maps created with multi-session mapping, where camera data and lidar data are used in separate sessions. The maximum depth of the scans and images are set to 15 m and 4 m, respectively.



(a) With loop closure. The maximum depth of the scans and images are set to 15 m and 4 m, respectively.



(b) Without loop closure. The maximum depth for the measurements is set to 4 m.

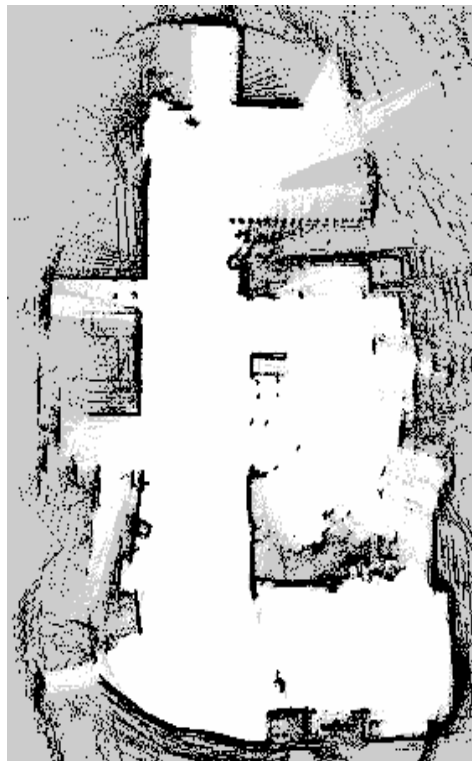
**Figure 7.14.:** Maps created with data from two cameras and two lidars fused into a point cloud.

### 7.2.3. Final Maps

The final maps used for navigation are shown in Figure 7.15, and are both created by Cartographer. The maps are created based on the configurations found throughout the tuning process. Figure 7.15a is created with data from the cameras and the lidars, while Figure 7.15b is created with scans from the SICK lidars solely. The latter is made to test the behavior of Navigation2 in areas where the map does not include obstacles above 150 mm.



(a) Sensor input: two scans and three point clouds.



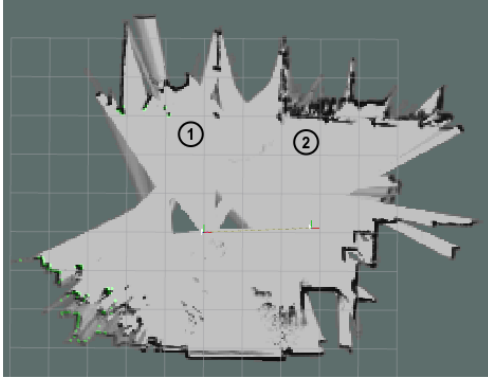
(b) Sensor input: two scans.

**Figure 7.15.:** The final maps created by using the Cartographer stack.

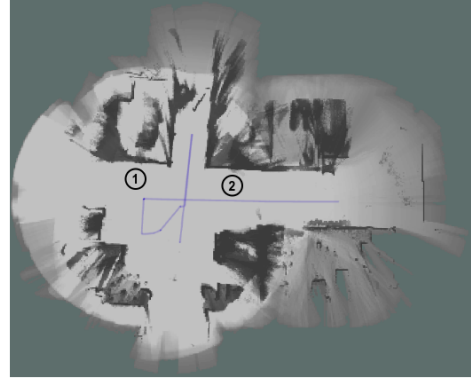
### 7.2.4. Mapping the Robot Cells

This result is included for a comparison with the previous work. A problem faced in that work was that the gratings around the robot cells were installed above the scan height of the SICK scanners. The previously created map is shown in Figure 7.16a. As the area could not be mapped successfully, this was a motivation for installing RGB-D cameras at the KMP. The gratings are to be lowered to 150

mm, but currently, this is only conducted for parts of the laboratory. Figure 7.16b shows the area mapped by Cartographer based on both scans and point clouds. Robot cell number one and two in the picture have gratings in height 180 mm and 150 mm above the ground, respectively.



(a) Created during the specialization project [3]. The sensor input is two scans.



(b) Created during the work of this thesis. The sensor input is two scans and three point clouds.

**Figure 7.16.:** Maps of the robot cells created by Cartographer.

### 7.3. Discussion

Different SLAM options for ROS2 were researched, which led to Cartographer and RTAB-Map. The requirements were that the stack needed to support both lidar and visual SLAM. Both chosen stacks make it possible to use all of the available sensor data and experiment with different sensor configurations. In general, few SLAM implementations for ROS2 were found during the research. Steve Macenski from Samsung Research America is the main developer of several ROS2 stacks, and according to him, does it not yet exist any satisfying SLAM implementation for ROS2 [20]. Macenski is currently working on developing the SLAM Toolbox, which is to be the default SLAM implementation of ROS2 [20]. In his repository, he states that both Cartographer and RTAB-Map have quirks that make them unusable for production robotics applications, and that Cartographer suffer from lack of maintenance [77]. SLAM Toolbox is not compatible with the desired use, as it is under development and currently only supports one lidar. This was also the case for another ROS2 library that was considered, namely LaMa, Alternative Localization and Mapping. The chosen sensor configurations were considered as the most relevant for each of the stacks. Both Cartographer and RTAB-Map contain a large number of parameters, which in theory can be

tuned for weeks. The obtained results are satisfying for discussing which stack and sensor configuration that provides the better maps.

### 7.3.1. Odometry

The lidar odometry, shown in Figure 7.4a and Figure 7.6a, is initially fair estimates. At the end of the hallway, both Cartographer and RTAB-Map faced challenges with the calculations. For Cartographer, this led to a big drift and poor results. For RTAB-Map the odometry was lost, and the execution was terminated. According to the documentation of RTAB-Map, the lidar odometry may drift if there are no constraints on the direction of the robot [22]. This typically occurs in hallways with parallel walls, such as the experimental area. In contrast to visual odometry, lidar odometry can not recover from being lost when there is no motion prediction. Considering Figure 7.4a, the motion prediction at the end of the hallway is not correct compared to the ground truth, which indicates that the odometry is lost.

The visual odometry, shown in Figure 7.4b and Figure 7.6b, did not suffer from drift at the end of the corridor. The walls of the hallway are textured, providing features for the matching process. While calculating the odometry, RTAB-Map provides information about the quality of the odometry. If the quality is above 100, the environment has enough features [78]. The observed quality during the experiment was between 200 and 400, which means the environment is good for visual odometry. A small drift occurred initially for Cartographer, which later was corrected. RTAB-Map provided a more wavy trajectory, which might have been caused by a depth estimation error. This error is addressed in the discussion in Section 7.3.2.

Figure 7.5 shows the trajectory estimated by Cartographer based on both lidar and visual odometry combined, which provided a reasonable trajectory compared to the ground truth. A small drift occurs in the same position as the lidar odometry got lost, which was corrected by the visual odometry.

The trajectory in Figure 7.3 is solely based on the encoders of the wheels, which is commonly referred to as erroneous. Visual and lidar odometry are applied to correct the error [79]. The trajectory in Figure 7.3 does, to a great extent, corresponds to the actual ground truth. In addition, the KMR robot is newly purchased and the wheels do not suffer from wear. Hence, it is reasonable to assume that odometry from the encoders is reliable.

### 7.3.2. Mapping

#### Cartographer

Cartographer contains parameters that can be tuned for configuring the applied algorithms and processing of the sensor inputs. The parameters related to range measurements apply for all inputs, and it is not possible to emphasize or process the different measurements separately.

As the configuration of the measurement depth applies to all of the input sources, the sensor data was not fully exploited for every case. When using only scans from the lidars, the range was set to 15 m. The layout of the MANULAB does not allow for sensor measurements with the original range of the SICK lasers, and a range of 15 m was specified to avoid uncertainties in the readings. When point clouds from the cameras were provided, the range for all sensors was set to 4 m. The maximum range of the cameras, given in Table 3.9, is approximately 10 m. For this range, the floor segmentation algorithm of Cartographer did not provide satisfactory results. Maps created with a longer range than 4 m contained areas of high uncertainty. Floor segmentation is not applied to the laser scans as the scans are parallel to the ground, and hence, longer ranges could be used. The `depthimage_to_laserscan` package used for converting the depth images to scans does not implement floor segmentation. If converting the whole image to a scan, the floor is considered as an obstacle. To handle this, the number of pixel rows in the image used to generate the laser scan was downgraded to crop the images in vertical direction. The maximum depth for the conversion is set to 4 m to ensure that the distant floor is not encountered.

Cartographer uses the pose extrapolator for estimating the position, which is based on a combination of odometry from the range sensors and the wheel encoders. Trusting odometry from range measurements led to a more jerky trajectory, while emphasizing the wheel encoders led to a straight path. On the other hand, weighting the range measurements low, corresponding to big uncertainty in the measurements, led to an incomplete map. The Ceres scan matcher was tuned to emphasize the odometry from the wheels as an initial guess more than the S2M match, according to the results obtained. The local SLAM results, when using only the SICK scanners, shown in Figure 7.7a, seem to be affected by the drift in odometry from lidar odometry. Scans contain less information than point clouds, making it more challenging to match scans than features. According to the documentation of Cartographer's lidar-based approach, scan matching accumulates error over time that needs to be corrected by graph optimization [80]. As seen in Figure 7.7b, the loop closure enhances the resulting map.

In the experiments conducted with point clouds from the camera, Figure 7.8 and 7.9, Cartographer provides the same results for local SLAM as with loop closure.



This is consistent with the results obtained in Section 7.2.1, where the use of camera data results in a good localization estimate. When using both scans and point clouds without loop closure, as in Figure 7.9a, significant obstacles only measured by the point clouds are not included. Increasing the number of range data per submap provided a more decent result, shown in Figure 7.9b. Separately, all of the generated submaps should include every obstacle, but a large number of overlapping submaps with different observations may lead to that perceptions disappears from the global map. This was also the case when the input was five scans, shown in Figure 7.10a. In Figure 7.10b, the size of the submaps was increased and loop closure was applied. This led to a slightly more detailed map, similar to the improvement shown for the map obtained with two scans and three point clouds. Drifts are observed for local SLAM, which was successfully corrected by the loop closure.

### RTAB-Map

The most significant tuning has been related to using multiple cameras, and also using both 2D lidar data and 3D image data. An essential parameter for making more of the obstacles be projected to the 2D map has been the *NoiseFilteringRadius*. By disabling this parameter, more obstacles are included as they are not removed as noise. As for Cartographer, only one depth can be defined for the range measurements. The same ranges, hence 15 m and 4 m, are used for the lidar and camera data.

When using two lidar scans, they must first be converted to point clouds, thereafter merged into a single cloud. RTAB-Map has a node called *PointCloudAggregator*, which is used for merging the point clouds. The result, shown in Figure 7.11, is a fair map, but, as expected, it lacks features above the lidar plane. Including loop closure creates a few more double walls, indicating that some loops are wrongly detected.

When using RTAB-Map with camera data, a problem related to depth estimation was discovered. Additional walls are created as the front camera and cameras on the sides calculate different depths to the walls. The depth estimated by the cameras is different when looking down the hallway and when looking directly at the wall. This could be related to the known computer vision problem of featureless walls [81, p. 83], but as the walls in the environment have features and the D435 cameras use an infrared texture pattern to avoid this problem, this seems unlikely. Our suggestion is that RTAB-Map is struggling with comparing the features seen by the different cameras as they are seen in entirely different angles and do not have overlapping fields of view. This is supported by the fact that the extra walls are later removed when the area is revisited. The additional

walls can easily be seen along the lower right wall in Figure 7.12. This area was only visited once, as the robot moved sideways on the way back and no camera re-observed the wall. These walls would probably be removed if the area was revisited. Except for the redundant walls, the maps provide a good recreation of the environment, and the objects lacking when using only lidars were present. Loop closure added some errors to the path, but did not affect the overall map.

Multi-session mapping makes it possible to initiate a new map with its own referential when starting up. When visiting previously explored areas, a transformation between the maps can be found, and the maps can be merged. This functionality is mostly used to combine multiple smaller maps of big floors with separate rooms. The functionality was tested to utilize both camera data and lidar data in the same map, by changing the parameter *Grid/FromDepth* between the sessions. This also makes it possible to change other parameters for the camera and lidar data, like the maximum depth for mapping.

As can be seen in Figure 7.13, the result of the multi-session mapping depends on the order of sensor configurations. As the sensors observe different surroundings, it seems like the environment has changed from the last mapping, and the last data will seem more likely. When using camera data in the second session, more features are added to the map, while if using lidar data in the second session, previously added features are removed. The final result is a merge of the maps created by cameras and lidars separately, where the last mapping session is weighted more. As previously seen, the loop closure did not improve the results, and hence, only the maps without loop closure are replicated. It is superfluous to map the whole area all over as there are only some features the lidars do not include. What would probably lead to a better result is to first make a map with lidar data, then do another mapping with the cameras, only focusing on areas with objects that the lidars did not catch.

Another point of improvement could be to simplify the path. As of now, the map created by the multi-session mapping includes a path past the same walls up to eight times. The more times the lidars pass an area that seems empty, the more confident the mapping becomes that this object is gone. If the lidars only scan the area once, it is more likely that the object would be left in the map. This was observed during the mapping, as the objects were not removed before the second time passing them.

The multi-session functionality gives decent results, but as Cartographer manages to utilize all the data in one run, it was desired to achieve the same with RTAB-Map. We considered the possibility of converting the 2D scans into point clouds, and then fusing them with the three point clouds from the cameras. This could be provided as a single laser scan point cloud input to RTAB-Map. When using the `pointcloud_to_laserscan` package to convert the laser scans, the fields of the

cloud is not equal to the fields of the camera point clouds. When using the `PointCloudAggregator` it is only possible to merge the clouds if they contain the same fields, or the same amount of points, where neither criteria are fulfilled here. To solve this, the laser scan clouds must be manipulated to equal the camera point clouds. However, the amount of work this requires caused this solution to be excluded.

Instead, a similar solution as for `Cartographer` was tested. The depth images of the cameras are converted to laser scans, by the same package as in `Cartographer`. Further, they are transformed back to point clouds as the original laser scans. This caused the point clouds to be similar, making it possible to merge them into one scan cloud. The maps created by this scan cloud is shown in Figure 7.14. It is desired to use 15 m range for lidars and 4 m range for the cameras. With this configuration, in Figure 7.14a, the lidar scans observe areas further away than the cameras, and remove obstacles only observed by the cameras. As the robot moves away from the area, and the object no longer is detected by the cameras with a shorter range, it seems like the object has disappeared from the scene. The loop closure also seemed to have trouble with matching data in different heights. This caused a map where objects were removed and with multiple double walls. When visualizing the data during the mapping, it was observed that the scans created from the depth image were more uneven and noisy than the lidar scans. The data from the cameras are going through multiple transformations, and this likely affects the data. Therefore, we are not fully satisfied with this configuration. The map in Figure 7.14b, where all sensors used 4 m range and without loop closure, was the best result achieved by the use of a scan cloud.

### 7.3.3. Remarks

A limitation of the results in Section 7.2, is that they are created from only one bag of data. This enhances the possibility for comparison between the maps, but noise and errors in the recordings may have affected the results. During tuning of the algorithms, a large number of experiments were conducted, leading to maps not included in the results. This also applies to other bags of data. Observations showed that the results were not always consistent. For instance, local SLAM with two scans could provide a perfect map in some cases and others not.

In the bag used to produce the maps in Section 7.2.2, the KMR revisited the same areas multiple times from different points of view. A simpler, non-overlapping trajectory leads to a less complicated mapping task, which probably could have resulted in more neat maps. On the other hand, according to the previous discussions, the maps could, in specific cases, benefit from more measurements.

A challenge with both `Cartographer` and `RTAB-Map` was to fuse the input data.

When the maximum depth of the scans is set higher than the value for the camera data, objects only captured by the cameras are removed. This was a pervasive problem and was often solved by reducing the range of the lidars or increasing the size of the submaps. The map of the robot cells, in Figure 7.16b, is based on point clouds and scans. The lidars only detect robot cell number two, and thus, robot cell number one is reproduced solely on data from the cameras. Thus, the installation of the 3D cameras was successful in order to map the robot cells.

Another observation made throughout the experiments was that when local SLAM was decent, loop closure tended to add more errors to the map rather than reducing them. Visual loop closure detection is not error-free, and very similar places can trigger invalid loop closure detections [22]. As previously discussed, the odometry from the wheel encoders does not provide significant drift and correction by global SLAM might not be necessary.

Of the configurations tested for Cartographer, the most successful map is the one in Figure 7.9b, created by using two lidars and three point clouds. For RTAB-Map, the better map was the one created by using multi-session mapping with lidars in the first session and cameras in the second session, shown in Figure 7.13b. As Cartographer is the more convenient option of the two stacks, it was chosen for creating the final maps shown in Figure 7.15.

# Chapter 8.

## Navigation

The Navigation2 stack has been used for navigation of the KMP. The objectives of this thesis state that the robot should be safe to navigate autonomously. Within safe navigation, we have chosen to emphasize dynamic replanning, obstacle avoidance, and the ability to adjust to the safety restrictions of the KMR.

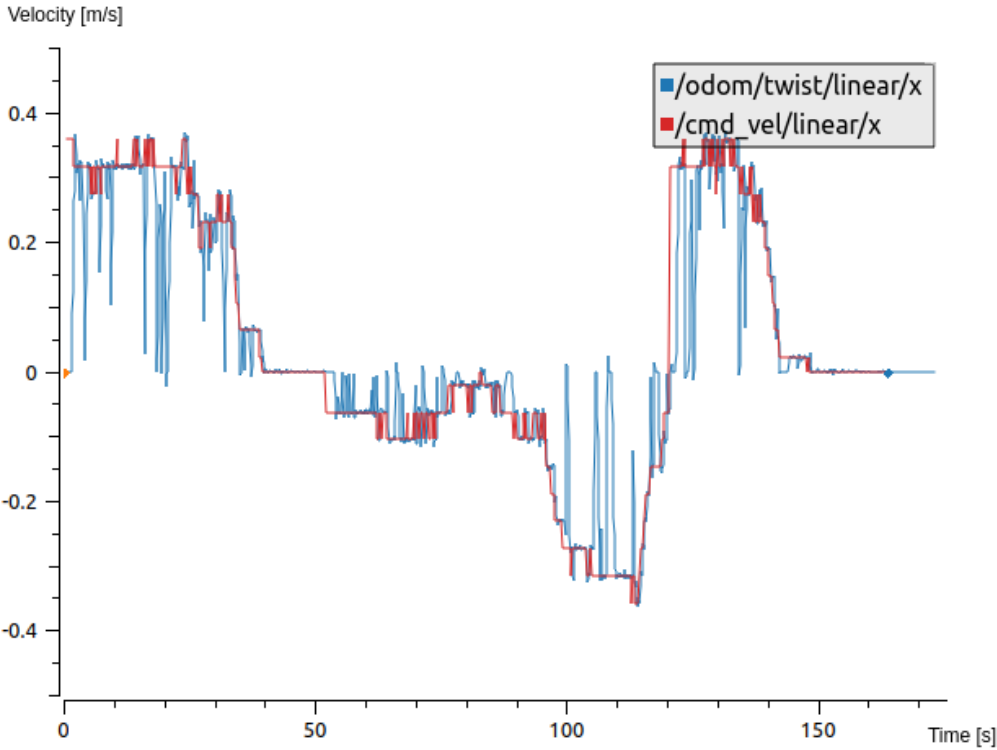
The experimental environment is the same as described in Section 7.1, and the map used is the one in Figure 7.15a. Section 8.1 presents the results from experimenting with different functionalities. Section 8.2 comprises a discussion of the tuning of the different algorithms applied in Navigation2, together with an evaluation of the obtained results.

### 8.1. Results

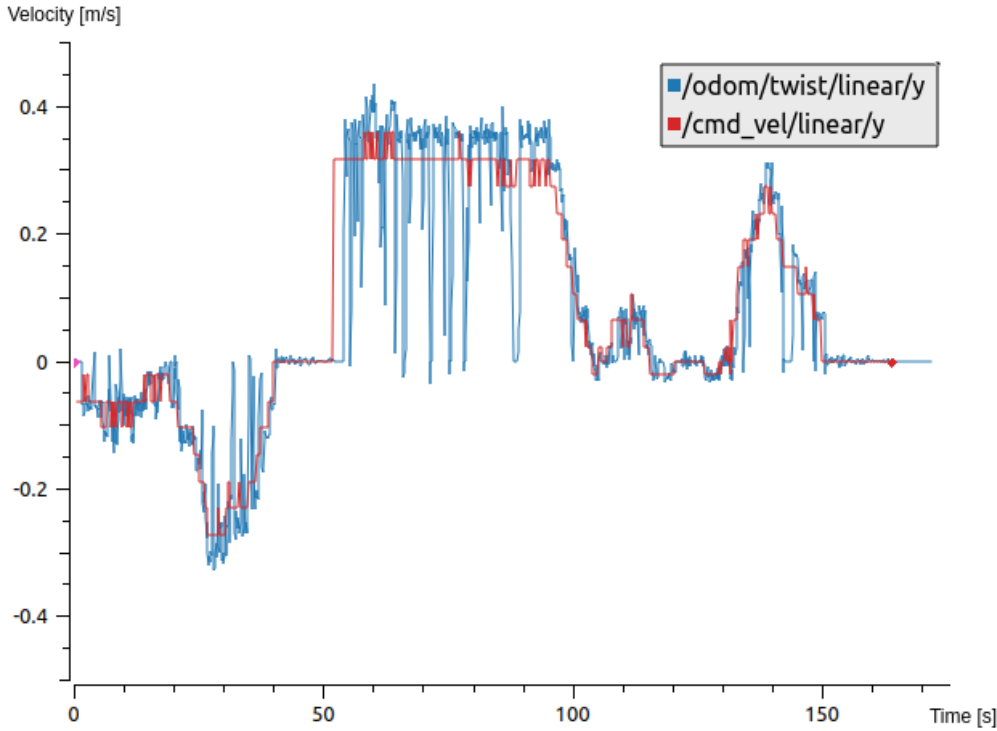
The experiments were carried out by providing an initial pose and goal pose in Rviz, and utilizing the communication architecture to send commands and retrieve sensor data. For these experiments, the communication nodes related to the KMP is applied.

#### 8.1.1. Velocities

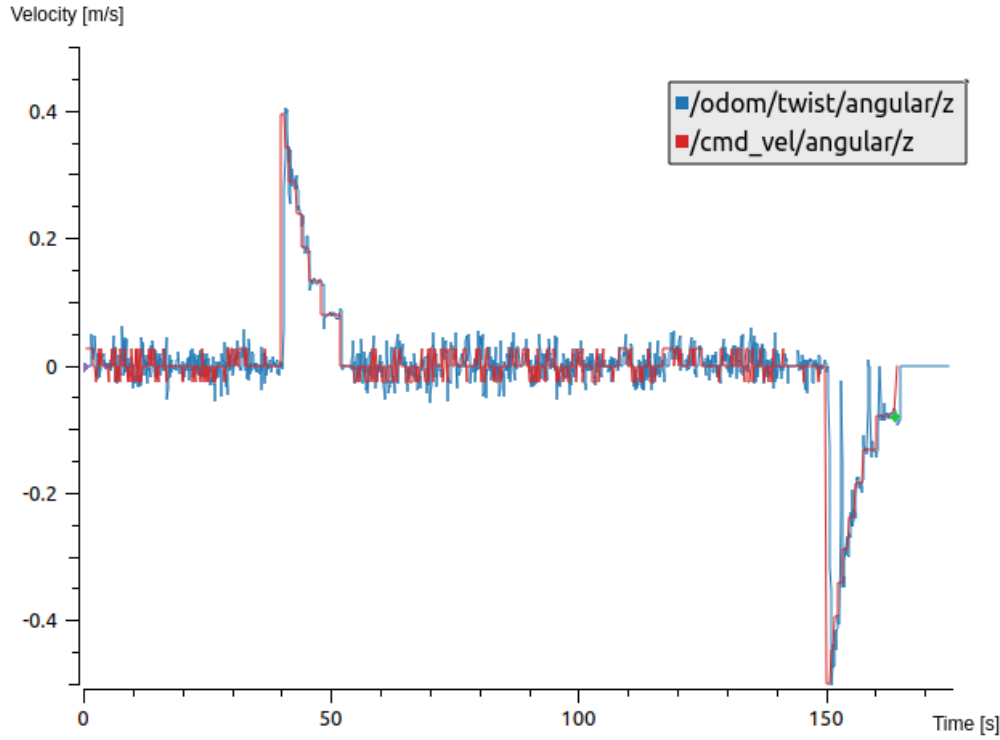
The first experiment is related to how the KMP follows the velocity from the controller in Navigation2. Figure 8.1 shows the odometry readings from the wheel encoders of the KMP in blue, together with the velocities commanded by the navigation controller in red.



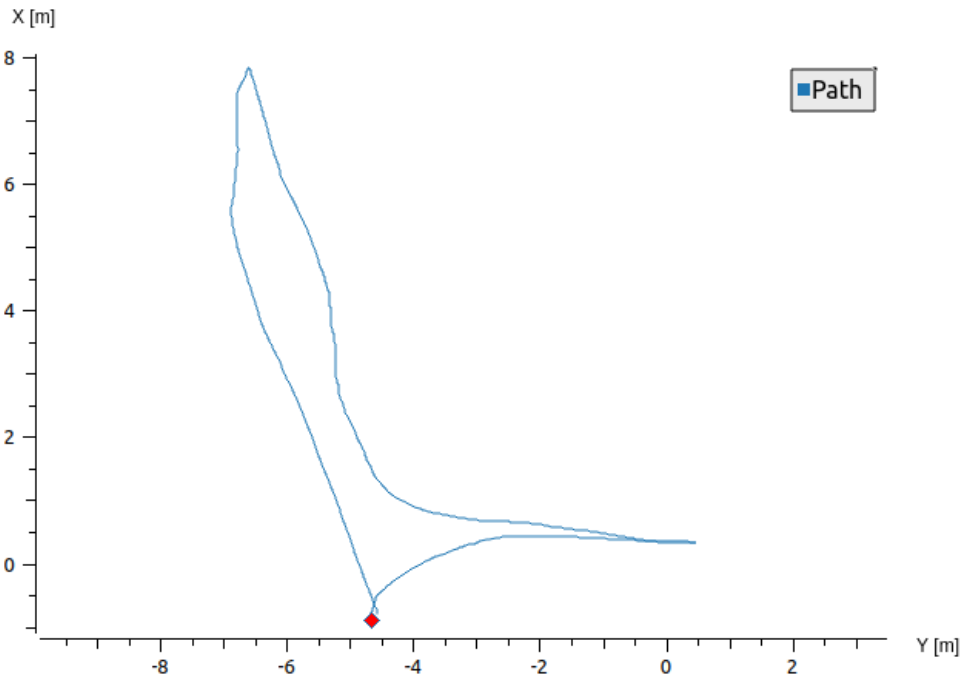
(a) Linear velocity in x-direction



(b) Linear velocity in y-direction



(c) Angular velocity



(d) The path followed by the KMP. The red square denotes the end of the path which is approximately where the path starts.

**Figure 8.1.:** The velocities retrieved from the wheel encoders of the KMP are plotted together with the commanded velocities generated by Navigation2. The path followed by the robot is shown in (d).

### 8.1.2. Dynamic Replanning

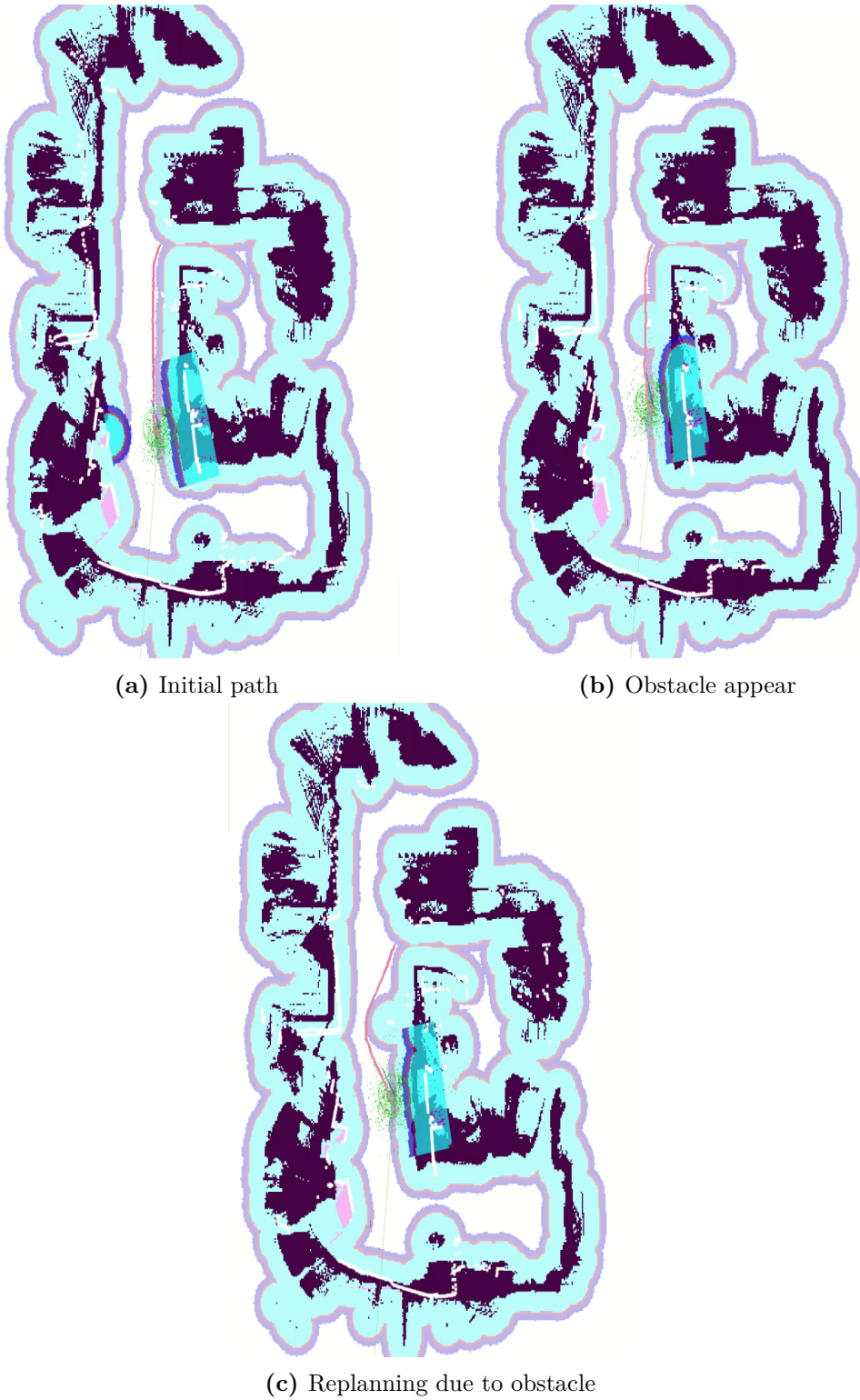
An essential feature of Navigation2 is the ability to replan when obstacles appear in the environment. For this to be possible, the communication must be sufficiently fast, as the controller needs to have enough time from the sensor data is retrieved to detect obstacles and create a new plan.

Figure 8.2 shows images of the environment during an experiment. The black areas are the occupied cells from the static occupancy grid map. The global costmap is composed of a static layer and an inflation layer. The static costmap is shown in blue color and the inflation layer in purple. The circular robot footprint is shown with a green circle, together with the also green particle cloud from AMCL. Sensor measurements of the environment are shown in white color. Around the robot, there is a local costmap, denoted with more defined blue and purple colors. The planned path is pink and connects the origin of the robot to a given goal pose.

When Navigation2 retrieves a goal pose, a path is created, and the robot starts moving. This is shown in Figure 8.2a. In Figure 8.2b, a person has stepped into the hallway, which can be recognized by a white spot where the sensors detect the person's legs and a costmap around the new obstacle. In Figure 8.2c, the global planner has taken the new obstacle into account and made a new global plan.

The attached video *replanning.mp4* shows the robot moving together with a recording of the Rviz window during the experiment. A box is placed in the hallway, and as it can be seen from the video, the robot drives around the box to avoid a collision.

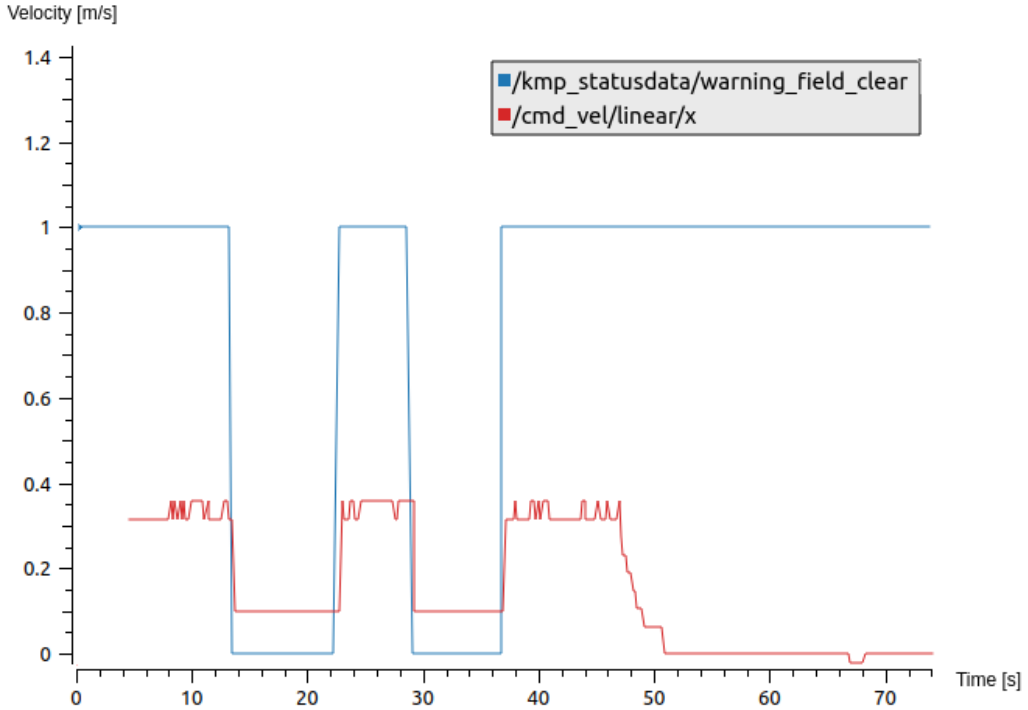




**Figure 8.2.:** Dynamic replanning with Navigation2. The images are screenshots from Rviz taken during the experiment. When an obstacle appears in the environment, a new global plan is dynamically calculated.

### 8.1.3. Dynamic Adjustment of Velocities

The NavigationSupport node is implemented to dynamically change the maximum velocities coming of the controller based on the status of the warning field. The experiments are performed by navigating down the hallway, while two objects are located near the path. Figure 8.3 shows how the velocity commands in x-direction sent from the controller are affected, based on if the warning field is violated or not. The blue line represents the boolean value of the warning field status, where the value 0 corresponds to a violation of the fields.



**Figure 8.3.:** The velocity commands from Navigation2 is dynamically adjusted based on whether the warning field is violated or not. A value of 1 denotes that the warning field is clear, while 0 indicates that an object is nearby.

### 8.1.4. Voxel Layer with Camera Data

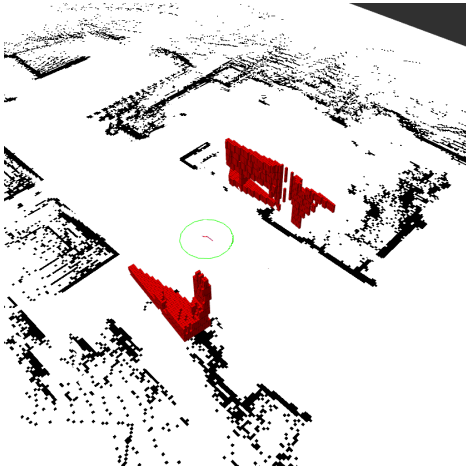
Point clouds can be used as input to generate a 3D obstacle layer in the costmaps by configuring the local and global planner with a voxel layer. This is useful for areas with obstacles in such a height that the lidar scanners can not detect them. Such an area is used in this experiment, and is shown in Figure 8.4. The map used in the experiments is shown in Figure 7.15b, and is solely created from lidar

scans. The lidars only detect the legs of the desks, and thus, the area in the map is clear.

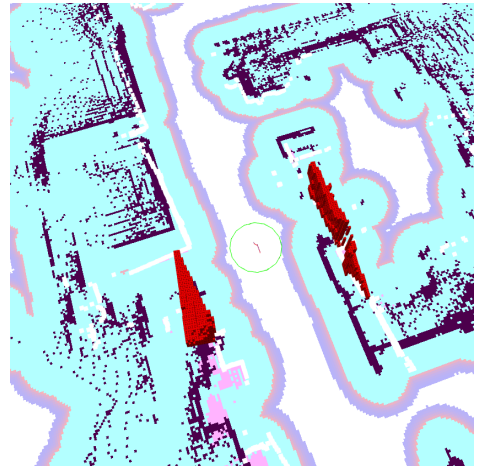


**Figure 8.4.:** Experimental area for testing the voxel layers with obstacles only detectable by the cameras.

In Figure 8.5a, it can be seen that the cameras detect the desks and denote the area as occupied. The costmap is affected as shown in Figure 8.5b.



(a) Static map



(b) Including costmap

**Figure 8.5.:** Maps with voxel markers in red denoting where the cameras detect obstacles.

## 8.2. Discussion

Much of the work comprising Navigation2 has been related to tuning of parameters. This process will be addressed in Section 8.2.1. The results presented in Section 8.1 will then be evaluated in Section 8.2.2.

### 8.2.1. Tuning

Navigation2 consists of external algorithms, where each has parameters that must be tuned. The default parameter file in Navigation2 is created for the demo robot Turtlebot3 and was used as a starting point. However, the Turtlebot3 is a smaller robot with different preconditions regarding drive type, velocities and footprint than the KMP.

Important parts of the navigation stack is the localization, performed by AMCL, planning, performed by Navfn planner, control, performed by DWB, and the costmaps used for the previous parts listed.

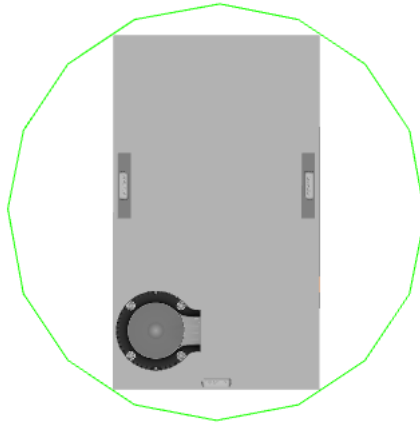
#### Localization

As a good initial pose of the robot always is provided, the algorithm has shown sufficient performance. Further tuning of AMCL has therefore not been required. It was observed during the experiments that if the provided initial pose was a bit off and the navigation started right away, the chance of hitting an obstacle increased considerably. AMCL uses some time to localize, making the uncertainty between the localization, map and observed obstacles high, leading the robot to drive close to obstacles. To make sure the robot is adequately localized, the robot should be manually driven a bit back and forth before starting navigation, such that the robot can be localized in the map. As we use the same starting pose each time, the exact coordinates are known and the pre-driving is not necessary.

#### Planning

The Navfn planner assumes a circular robot, which is a disadvantage. The footprint of the robot must hence be circular, and big enough to cover the whole robot. This creates a footprint as in Figure 8.6, which is bigger than the actual robot.

A global path for the robot is only created if the robot can drive from start to goal without hitting any obstacles. The circular footprint limits the possibilities for the robot to drive in narrow places where the robot, in reality, would be able to drive with the front first. This is also a problem when approaching work stations.



**Figure 8.6.:** As the Navfn planner assumes a circular robot, the circular footprint of the robot must be big enough to cover the corners of the robot.

The planner assumes the robot is wider than it is, which makes it hard to stop within a reasonable distance from the work stations.

Compared to the other algorithms used in the navigation stack, the Navfn planner has few tuneable parameters. This makes it difficult to change the behavior to fit our needs in a better way. This issue is highly relevant for other users of Navigation2, and it is planned to implement a better planning approach that considers the actual footprint and dynamic abilities during 2020 [82].

## Control

The DWB local planner has, together with the costmaps, been the priority of the tuning.

The parameters `xy_goal_tolerance` and `yaw_goal_tolerance` define how close to the requested goal pose the robot must be for the navigation to succeed. The parameters are set to be 0.2 m and 0.2 radians, respectively. Both values were lowered from the default values as it is required that the KMP stops in the correct pose close to the work stations. The manipulator has a range of 820 mm, and would be able to pick up an object even though the robot is a bit further away. However, the camera mounted at the manipulator requires a decent angle for the object detection component to localize the object. Hence, a stricter measurement is specified for the angle.

Another important parameter is the `sim_time`. When the DWB algorithm search for local paths, the possible velocities are simulated to evaluate where the robot

would end if this velocity was applied for the given `sim_time`. This parameter affects the choice of local paths, and hence, the behavior of the robot. If the value is low, i.e., 1 second, it is observed that the robot struggles with reaching the goal when it is close to the right position. It seems like the time is insufficient to obtain a trajectory that actually reaches the goal. The controller frequency is set to be 5 Hz, which means the local planner attempts to perform active replanning in time intervals of 0.2 seconds. When increasing the `sim_time`, the number of computations which must be performed in every time interval increases, and will slow down the controller. Hence, the value should be as low as possible, and at the same time be high enough for the planner to create continuous paths. We found that a value of 3 seconds was sufficient for our system.

To detect if the robot makes sufficient progress, the controller continuously checks if the robot has moved a specific distance within a certain time. The time and distance are controlled by the parameters `movement_time_allowance` and `required_movement_radius`. As the KMP moves holonomic, it performs mostly linear movements before finishing off with pure rotational movements to the correct orientation. When performing only rotations, the movement radius does not increase. Hence, the navigation would fail if the robot uses more time to rotate to the goal than the time allowed. The robot will, at maximum, rotate 180 degrees, or 1.57 radians. When moving at slow speed, the robot has a rotational velocity of around 0.1 radians, which means it would use about 16 seconds to perform half a revolution. To make sure the robot does not get stuck when rotating to goal, the movement time allowance parameter was set to 30 seconds.

The KMP has a higher maximum velocity in x-direction, corresponding to forward or backward motion. Even though, the maximum velocities for the controller was set equal for all directions. This makes the local planner not prioritize one direction over another. We chose to set the maximum velocity in x- and y-direction to 0.4 m/s, which is about a third of the available x-velocity. Low velocities decrease the chance of collisions, as it is possible to stop the robot if faults occur in the system.

The DWB planner choose the local path based on the score given by the different critics. Tuning the weight of the critics is crucial to control the behavior of the robot, and the weights are determined through multiple experiments in both simulation and the real environment. The critics are given the following weights, and an explanation is given in the following paragraphs.

- Twirling: 30.0
- RotateToGoal: 20.0
- PathDist: 20.0

- GoalDist: 32.0
- PathAlign: 0.0
- GoalAlign: 0.0
- BaseObstacle: 20.0
- ObstacleFootprint: 0.0
- Oscillation: 0.0

The Twirling critic is significant for holonomic robots. It penalizes rotations and makes the robot drive linearly as far as possible. Without this critic, the robot behaves mostly as a non-holonomic robot.

The RotateToGoal critic defines the behavior close to the goal. If decreasing the weight, all types of motions are accepted near the goal. This causes the robot to both rotate and drive back and forth in place. By increasing the weight, only rotations are allowed within a given distance of the goal. This makes the robot drive to the correct position, and rotate until the desired angle is reached. This creates more accurate paths and movements.

The critics PathDist, GoalDist, GoalAlign and PathAlign affect the robot in similar manners. As long as some of the critics are weighted, the robot follows the global path approximately and heads for the goal. If none of the critics are weighted, the robot drives completely off track, and does not seem to try reaching the goal. Whether to weight the PathDist or GoalDist more, depends on how much to trust the global path. Experiments show that having a high GoalDist weight often leads the local planner to cut corners, as it searches for the shortest path to the goal. The global planner depends on the global costmap, and hence, as long as the tuning of the global costmap is sufficient, the global path can be trusted. Therefore, the PathDist is weighted higher than the GoalDist. Discussions online revealed that the GoalAlign and PathAlign critics might lead to poor obstacle avoidance [83]. The critics were disabled as no improvements in the navigation were observed with the critics.

Avoiding obstacles is a high priority, and hence the obstacle critics, BaseObstacle and ObstacleFootprint, are important. It is desired that the robot rather takes a detour and avoid obstacles, than taking the fastest way to the goal. Through experiments, it was found that there are multiple methods to avoid obstacles, but combining all of them makes the path planning difficult. When using both the ObstacleFootprint critic and an inflation layer, the possible paths are limited as the footprint is not allowed to be close to the inflation layer. As the BaseObstacle critic makes sure the path does not go within the costmaps, and the inflation layer is tuned to be bigger than the robot, the combination of the BaseObstacle critic

and the inflation layer was sufficient for obstacle avoidance. The Oscillation critic was disabled, as oscillations never occurred as a problem.

## Costmaps

All of the available layers, which are the static, obstacle, voxel and inflation layer, are used for the costmaps. The obstacle layer is configured to use scan data from both SICK sensors, while the voxel layer uses point cloud data from the cameras at the mobile base. The sensor inputs are used for both marking obstacles in the map, as well as clearing them.

Deactivation of the built-in safety functions of the robot entails fully trusting the controller of Navigation2. The developers of Navigation2 recommend taking safety precautions when using the stack, as it is under substantial development. This makes the obstacle avoidance very important, and tuning the inflation layer enough has been demanding. In this context, the two most essential parameters are the `inflation_radius` and the `cost_scaling_factor`. The inflation radius decides how far away from the obstacle a zero cost cell should appear, while the cost scaling factor determines the slope of the cell values. If this parameter is increased, the cost is decreased. As it is desired that the robot avoids obstacles as much as possible, the inflation radius should be high. If the radius is too high, tight areas could be hard to pass through. A high inflation radius should therefore be combined with a high cost scaling factor. The area used for experiments consists of narrow areas, which limits the radius of the inflation layer to be 0.9 m. The cost scaling factor is set to be 10. This values creates the costmap shown in Figure 8.2.

Again, it is a disadvantage that the navigation stack assumes a circular robot. The circular footprint makes it a requirement that the robot can also pass through areas sideways or diagonally. This limits the possible paths and makes the tuning of the inflation layer difficult, as the radius of the inflation layer must be lowered to allow the robot to pass narrow areas.

Being strict on obstacle avoidance makes it harder to navigate close to work stations. The inflation layer limits how close to the tables the robot might drive. As good obstacle avoidance, with a high clearing when passing obstacles, is considered crucial, we decided to keep the inflation radius higher than desired for navigating to work stations, and instead find another solution to this problem.

## Recovery Behavior

When performing experiments with the real robot, the recovery behavior in Navigation2 was disabled. Information within the Navigation2 repository states that



the recovery behavior is under development, and collision avoidance is not integrated. It should therefore only be used during simulations.

### 8.2.2. Evaluation of Results

#### Velocities

As can be seen from Figure 8.1, the odometry of the robot follows the commanded velocity from the Navigation2 controller to a large extent. The odometry is more noisy than the commanded velocities, which may be related to the frequency of publishing. The odometry was initially sent from the robot and published to ROS at a frequency of around 90 Hz. This is unnecessarily fast, and amounts of data is dropped as Navigation2 can not keep up with the incoming data. To reduce the amount of unnecessary data transmitted, the frequency was lowered to 20 Hz, which is the same frequency as for the lidar data. The controller frequency is 5 Hz. Even though, as the path following was sufficient during the experiments, no further experiments with the goal of increasing the controller frequency were conducted. As the sensor data is available at a much higher rate, the frequency of the controller should ideally be higher. This depend on the available CPU, but also tuneable parameters which require more of the CPU. For example, the number of velocity samples and the simulation time used by the local planner when searching for available trajectories will highly affect how fast the local planner will find the best trajectory. The KMP is a large robot with holonomic drive, and require these parameters to be higher than for a smaller, differential drive robot.

As discussed in Section 6.2.2, the robot sometimes perform jerky motions. This can be seen in Figure 8.1, as the graph have spikes going towards zero. Only the wider spikes correspond to visible chopping in the motion, such as the spike that occur after 90 seconds, where the odometry remains at zero for a few seconds.

#### Dynamic Replanning

Figure 8.2 visualizes how the planner is able to perform dynamic replanning. This is an crucial feature when working in an environment together with other robots or humans. As can be seen from the video attachment, the replanning happens almost immediately after the obstacle is detected. During these experiments, the person appeared about 1.5 m in front of the robot. This gave the controller enough time to perform a successful replanning, and a shorter distance would probably also succeed. However, this was not tested due to safety reasons. As the lidar scanners have a range of about 30 m, insufficient time for replanning should never occur as a problem.

## Dynamic Adjustment of Velocities

The NavigationSupport node is implemented to create a similar behavior as the built-in safety system of the KMR. Figure 8.3 shows how the velocity commands in the x-direction from the navigation controller coincides with the status of the warning field. If the warning field is clear, corresponding to the value 1, the velocities are between 0.3 and 0.4 m/s, where 0.4 m/s is defined to be the maximum allowed velocity. When the warning field is violated, the maximum velocity is reduced to 0.1 m/s. Towards the end of the plot, the navigation has reached its target, and the velocities drop regardless of the status of the warning field. This functionality works as desired and makes it safer to use the robot in a real environment. It will also be safer for humans to work around the robot, as high velocities and sudden movements are prevented in the vicinity of humans.

During these experiments, the trigger of safety stops when the protective field is violated was disabled. If this functionality was active, it would be crucial to reduce the velocity when moving closer to obstacles to avoid emergency stops. When the velocity is reduced, so is the size of the protective field, which makes it possible to move closer to objects. As the safety stops should be enabled in further work, it is required that this functionality is present.

## Voxel Layer with Camera Data

Figure 8.5 shows how including camera data to the navigation improve the costmap, and hence the planning, within incomplete maps. This is useful in an environment with tables and objects in different heights, like the MANULAB.

The drawback of adding cameras is the increased amount of data to be processed and transferred over the network. As described in Section 6.1.2, launching programs and sending more data heavily affected the response time. When the delay is high, the navigation is affected as odometry and sensor data is not retrieved fast enough. As a result, it is not always possible to use three cameras for navigation. A possible solution is only to use one camera and only provide 3D measurements for the front of the robot. On the other hand, as the robot moves holonomic, it can not be assumed that the robot drives in the forward direction, and hence, using one camera is not a proper solution.

The importance of using 3D data for navigation depends on the map of the environment. A map could be created by using only lidar scans, and then both cameras and scans should be used for navigation. This solution would be better equipped for changes in the environment. If it is not possible to use camera data for navigation, a more detailed map created with 3D data is required. As the maps created with both lidar and camera data provide a good representation of

the environment, using cameras for navigation is not critical in our system. However, it is desired to utilize all of the sensors, but this must be evaluated based on the quality of the network.

### 8.2.3. Remarks

Overall, using the Navigation2 stack for navigating the KMP has shown to be a good solution. A demanding challenge was to tune the parameters to avoid obstacles and, at the same time, be able to get sufficiently close to work stations. The success ratio was increased by always making the robot turn with the front-facing the table, as there is less excess space inside the footprint at the front. For approaching work stations sideways, a solution could be to move the robot closer to the work station after the navigation has succeeded. This approach could be automated by always moving sideways for a specified distance to approach the work stations. If there were an obstacle between the robot and the work station, the navigation controller would not be able to navigate to the original goal pose due to the obstacle layer in the costmap. Hence, this is evaluated to be a safe solution. Another solution could be to exploit the cameras at the base together with computer vision to find the distance to the work station. This solution would be more generic as it would be applicable in every environment.

During the work with Navigation2, sudden crashes have occurred for the global planner and controller. This made navigation impossible and dangerous. When the controller stops working, the robot keeps moving with the last commanded velocity without functionality for obstacle avoidance. Much effort was put into figuring out this problem, which turned out to be a bug in rclcpp, the ROS Client library for C++. The problem was related to the rcl clock, which was not thread-safe. Rcl implements the common functionality for the language-specific ROS Client Libraries. When both the rclcpp clock and rclcpp timerbase access the rcl clock API without locking, this causes an internal state corruption on concurrent access in multi-threaded applications. The problem was solved by using a different branch of rclcpp, where a global mutex gets locked every time the underlying rcl clock structure is manipulated. This branch is a temporary solution to the bug for ROS Eloquent, as the problem is fixed for the next ROS release, ROS Foxy.



# Chapter 9.

## Manipulation

The MoveIt2 stack has been applied to perform motion planning for the LBR. The experimental setup is presented in Section 9.1. The results are presented in Section 9.2, and further discussed in Section 9.3.

### 9.1. Experimental Setup

The KMR was placed in front of a work station made out of stacked cardboard boxes, as shown in Figure 9.1a.

The experiments were carried out by executing a behavior tree and utilizing the communication architecture to send commands. Only the communication nodes related to the LBR are used. The sequence defined in the tree is to plan a path and move to a goal pose, then move back to the initial pose. The same initial pose and goal pose were specified in all of the experiments. Figure 9.1b shows the joint configurations of the start pose, together with the coordinate system used for the following results. The red axis denotes the x-direction, the green axis is the y-direction, and the blue axis is the z-direction.

### 9.2. Results

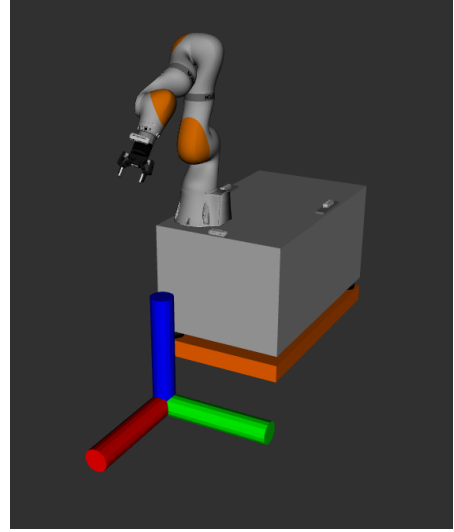
The experiment described in section 9.1 was performed three times, leading to MoveIt planning three different paths.

#### 9.2.1. Joint States

Figure 9.2 shows the joint states of the LBR together with the planned joint states in the trajectory from MoveIt for the three experiments. The joint states



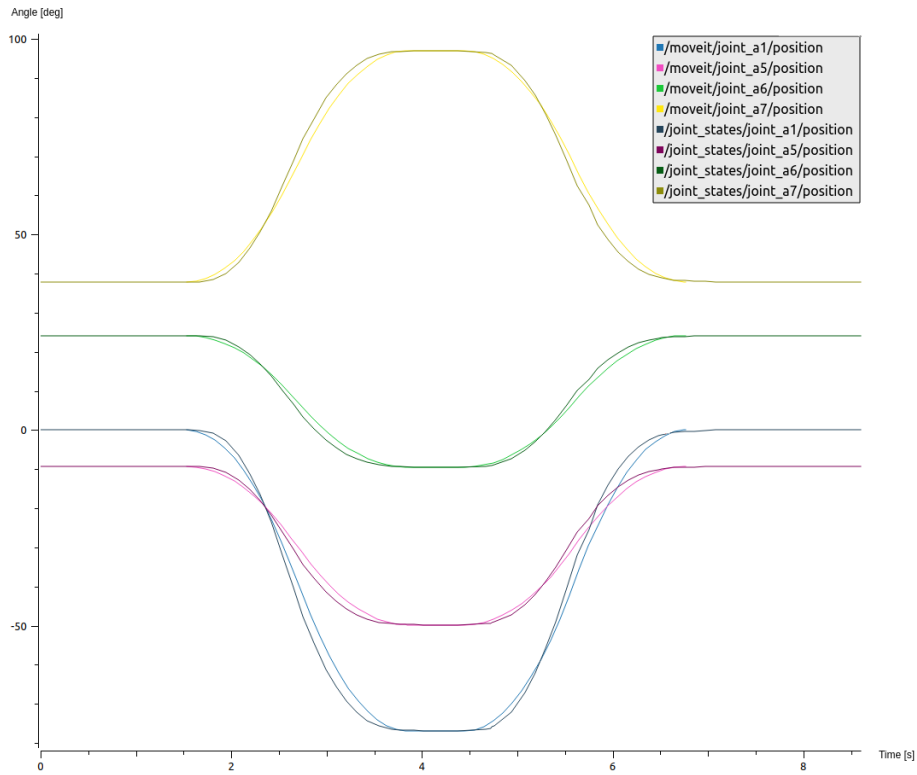
(a) The robot is placed in front of stacked cardboard boxes. The gripper middle point and the goal pose is denoted by the green and red markers, respectively.



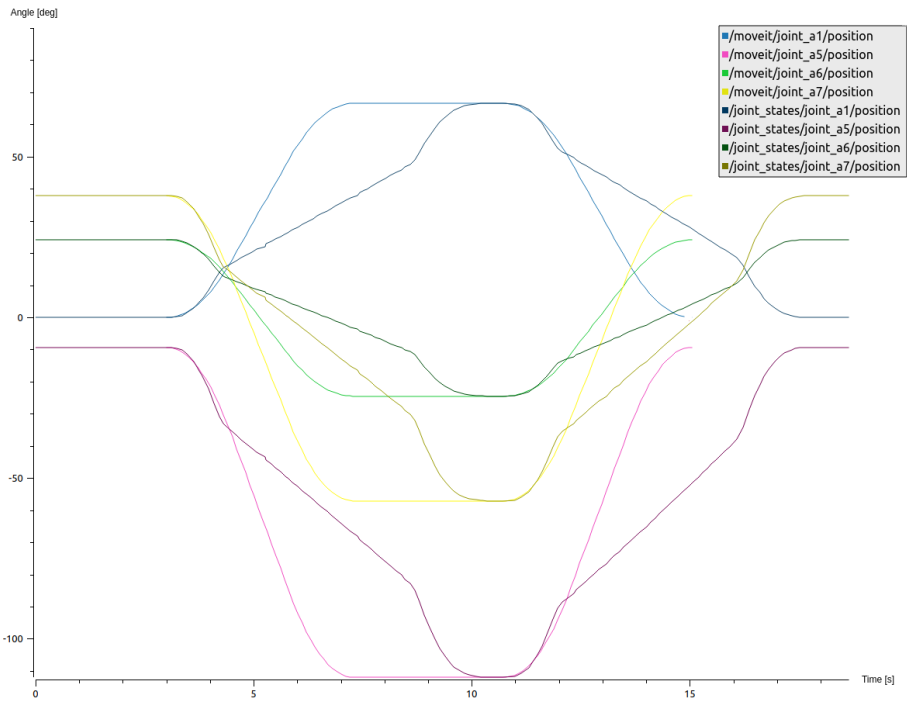
(b) The joint configurations for the start point during the experiment, together with the coordinate system used for the results.

**Figure 9.1.:** Experimental setup for manipulation.

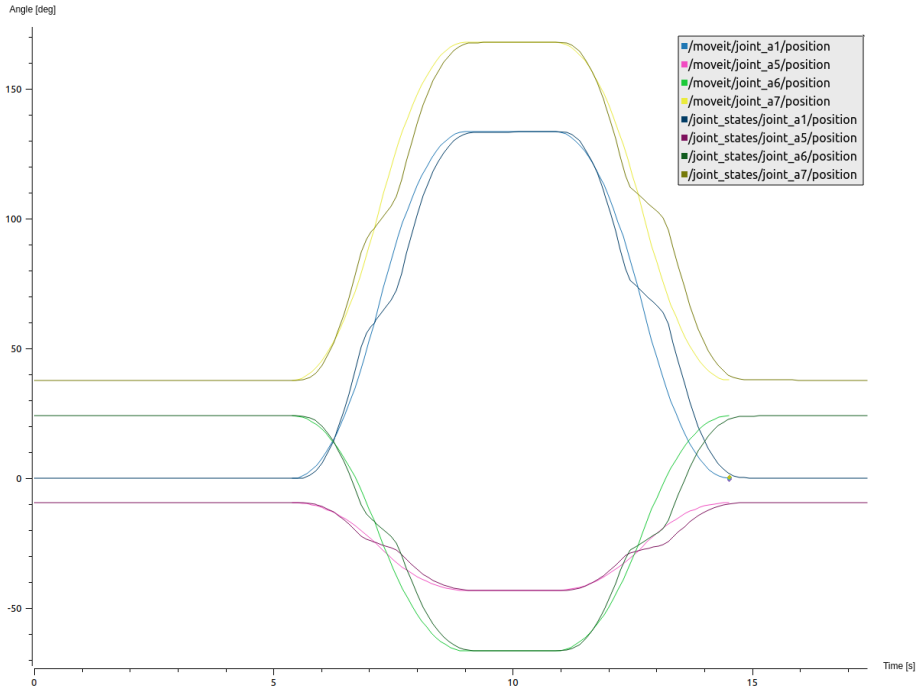
are shown for joint numbers A1, A5, A6 and A7, where the darker lines correspond to the actual states. The joint state values are relative to the configuration where all the joints are zero, which is when the manipulator is fully stretched upwards.



(a) Experiment 1



(b) Experiment 2



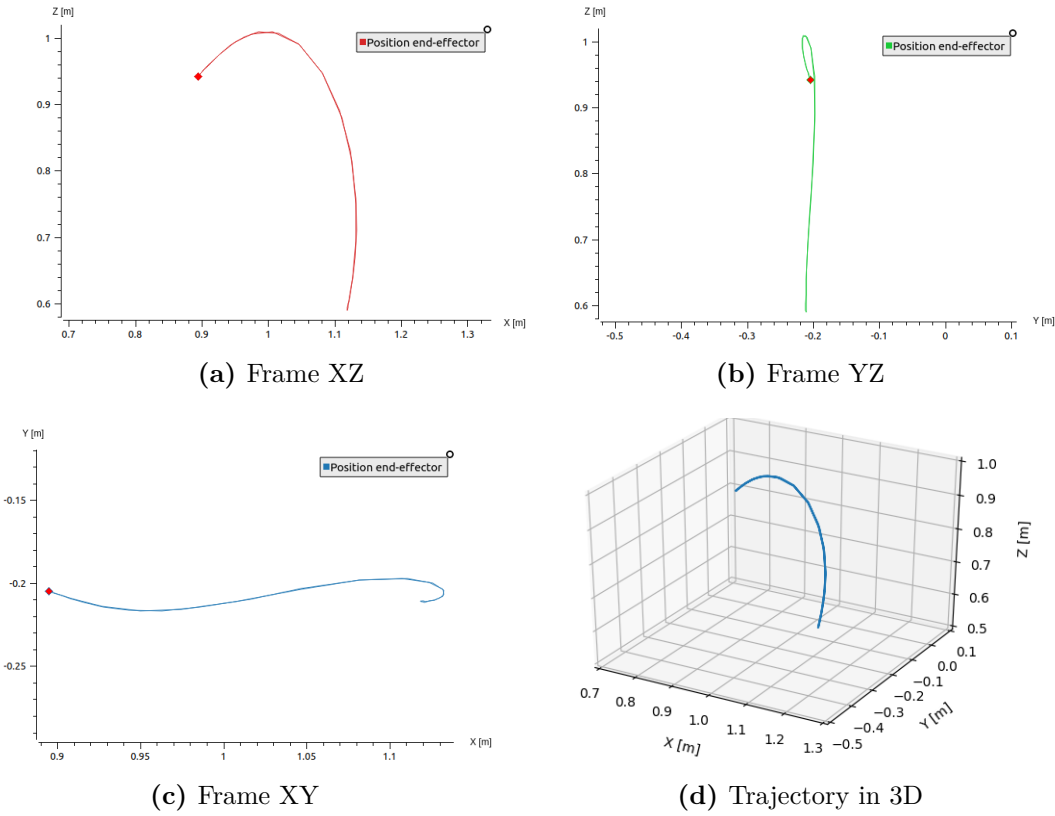
(c) Experiment 3

**Figure 9.2.:** The joint states retrieved from the sensors of the LBR are plotted together with the trajectory generated by MoveIt for joint number A1, A5, A6 and A7.

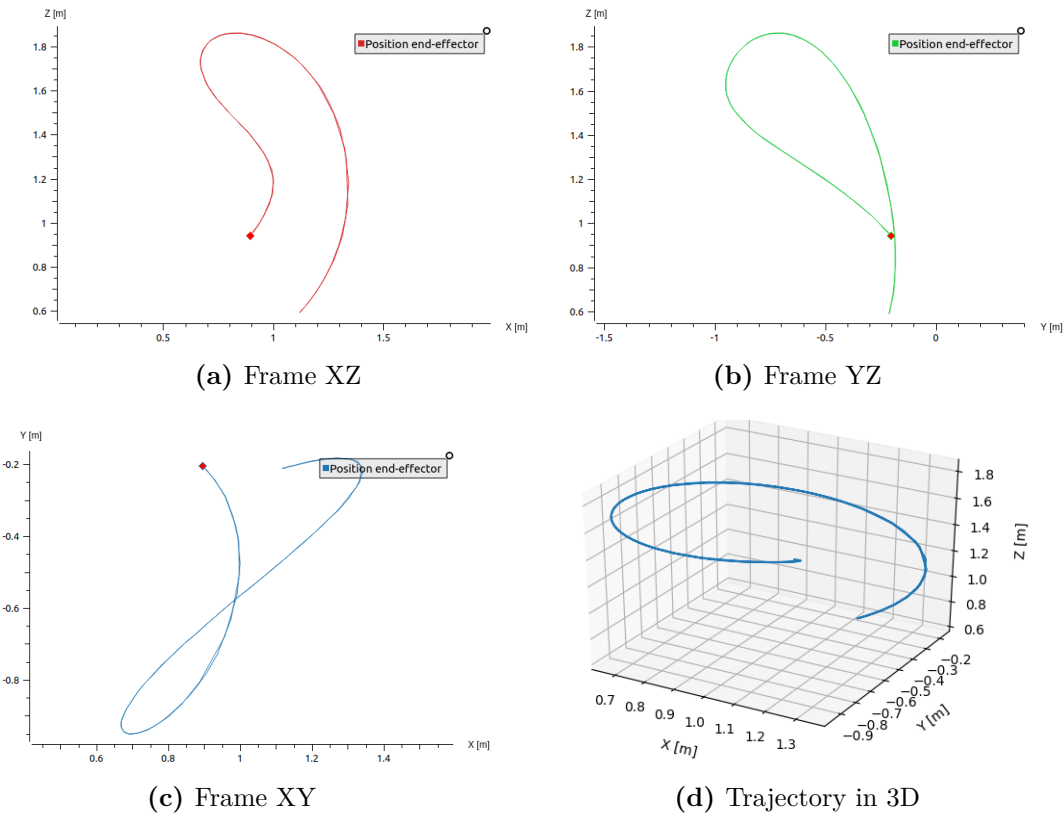
### 9.2.2. Path Planning

Figures 9.3 to 9.5 show the path of the middle point between the gripper's fingers for the three experiments. The results are obtained based on the transformation from the base frame to the center point calculated by the Tf package.

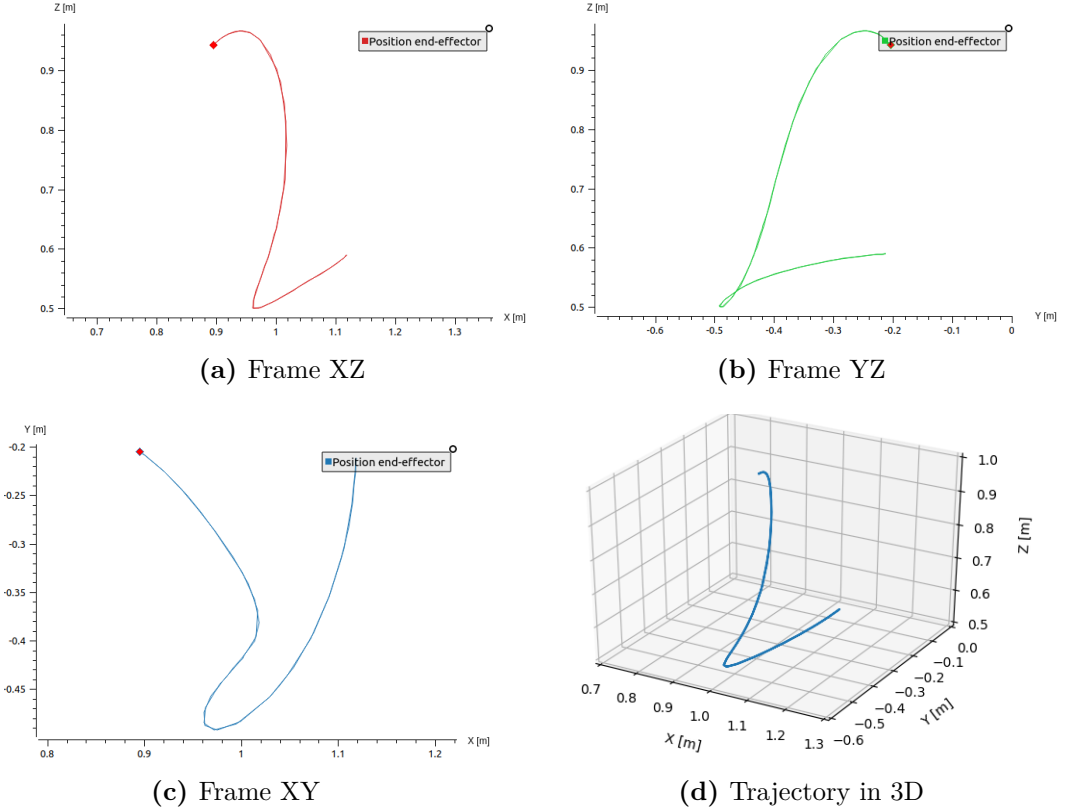




**Figure 9.3.:** The position of the end-effector during experiment 1.



**Figure 9.4.:** The position of the end-effector during experiment 2.



**Figure 9.5.:** The position of the end-effector during experiment 3.

## 9.3. Discussion

### 9.3.1. Hand Eye Calibration

The goal pose provided to MoveIt is defined in the optical frame of the camera mounted at the manipulator. A precise transformation between this frame and the middle point between the gripper fingers is required to create an accurate path. The hand eye calibration problem is to find the transformation between a sensor mounted on the robot actuator and the actuator itself. The ROS2 Grasp Library includes algorithms for conducting the estimation. The tools were applied, but the resulting transformation was not satisfactory, and the solution was aborted. The manipulator and gripper models in the URDF model are based on accurate representations. As we designed the adapter between the two components, the correct dimensions are known. The transformation between the optical frame and the base link of the camera is provided by the Realsense node and is also assumed to be correct. Through experiments, it was found that the transformation

obtained from the URDF model was sufficient, and that hand eye calibration was not necessary.

### 9.3.2. Joint States

As can be seen from the three plots in Figure 9.2, the joint states of the LBR follow the path obtained from MoveIt by varying quality.

In Figure 9.2a, the trajectory followed by the manipulator is, to a great extent, aligned with the planned trajectory. Common for all the joint states is that they have a bit steeper slope than the path from MoveIt. As the time of execution is short, the time difference for completion between the joint states and the path is less than half a second. As the execution is not happening in real-time, the difference in time is considered acceptable.

As the LBR moves slightly faster than the planned trajectory, this is assumed to be due to the approximate positioning performed when Spline motions are executed. According to the manual, slight path deviations may occur when a tool is attached to the manipulator [25, p. 370]. Another explanation of the deviation is that the LBR is not aware of the mounted components. This is a consequence of the gripper not being integrated into the Sunrise system. Hence, the weight of the installed components is not taken into consideration by the robot controller.

As can be seen from Figure 9.2b and 9.2c, the manipulator manages to follow the joint states of the trajectory, but uses a longer time to achieve the same states. This indicates a lower velocity than the trajectory. A reduction of velocity may occur for Spline motions with tight corners, due to abrupt change in direction, or for a major reorientation. During planning, these constraints are taken into consideration by the robot controller. The LBR moves as fast as it is allowed within the programmed velocity. The path in experiments 2 and 3 were more winding than the path in experiment 1. This can be seen from the corresponding end-effector trajectories in Figures 9.3d, 9.4d and 9.5d. This may have caused the observed reductions in velocity.

### 9.3.3. Path Planning

Figures 9.3 to 9.5 show the path of the end-effector from three experiments between the same two poses. As can be seen, the path between two poses may be very different for each new planning. The paths are examples of what was both desired and challenging when using MoveIt for motion planning. The path achieved in Figure 9.3 is what is desired. The path is almost optimal in terms of both the length of the path and the time of execution. The end-effector is moved in an approximately straight line down to the goal pose.

Figure 9.4 shows a path where the manipulator makes a big detour before ending in the correct pose. The subplots for frame XZ and YZ show that the position is almost similar for the start and end pose in both x- and y-direction. Even though, the subplot for frame XY shows that the end-effector point has moved a lot in both directions. OMPL, the default planner applied for MoveIt, is a library for randomized, non-optimized motion planning algorithms. As the paths are not optimized, this may lead to long, winding trajectories. This is acceptable for open surroundings, but may lead to hazard situations if people or obstacles are nearby.

Many planners in OMPL favor the speed of finding a solution over path quality [84]. The motion planner is configured by plugins, which means trying out another plugin could be a good option. Currently, no other motion planning libraries are ported to ROS2, but they will probably be soon. An example of a motion planner from MoveIt1 which could be used is the Stochastic Trajectory Optimization for Motion Planning, STOMP, which is optimization-based.

The path in Figure 9.5 reveals another problem. Both subplots for frame XZ and frame YZ show that the end-effector point goes low in z-direction before it is raised to the final position. As the manipulator should grasp objects from a work station, this path causes the gripper to collide with the work station. Another remark, which can be seen from the subplot of frame YZ, is that the gripper moves sideways, in y-direction, towards the goal. The orientation of the gripper is such a way that the gripper pushes the object off the table and away from its original position. Thus, there is no object to grasp. This problem could have been solved by setting the goal pose in a given height above the object, then performing a linear motion down towards the object. The linear motion could be performed by using a LIN-motion from Sunrise or the Cartesian Path Planner plugin for MoveIt. The latter is desired as the control should be performed through ROS.

The problem of not avoiding the work station is related to the lack of collision checking with the environment. As the manipulator was included into the system during this spring, the focus area has been to make the LBR compatible with MoveIt. MoveIt does have functionality that was not prioritized to implement, but which should further be implemented as the systems are compatible. This includes making a proper representation of the environment for collision checking. Either point cloud or depth image data from the camera at the manipulator could be used to update the planning scene dynamically. This would help the path planning to avoid the work station or other objects around the object to grasp. Another option is to use octomaps, which are 3D maps, to represent the environment. The octomaps can be directly used in the collision checking. RTAB-Map has functionality to create octomaps, and this could be utilized for creating maps to use with MoveIt.

### 9.3.4. Remarks

Using MoveIt for path planning gives the desired functionality. As the current version of MoveIt2 is the first beta version and was launched in February, functionality from the first version is lacking. New functionality is continuously ported, which should be utilized when it is ready. This especially applies to other motion planners.

Even if collision checking is enabled, there would be a problem if obstacles enter the area after the path is planned. In MoveIt1, there is support for dynamic path planning, but this is not yet available for MoveIt2. The only option is to plan a path, then execute it. A possible solution would be to monitor the area, and if an obstacle appears, the execution should be stopped, and a new plan should be created. This can be accomplished by utilizing parallel behavior tree nodes.

MoveIt works with our system, but multiple points of improvement are highlighted. These points do not require much effort and can be easily fixed in further work.

# Chapter 10.

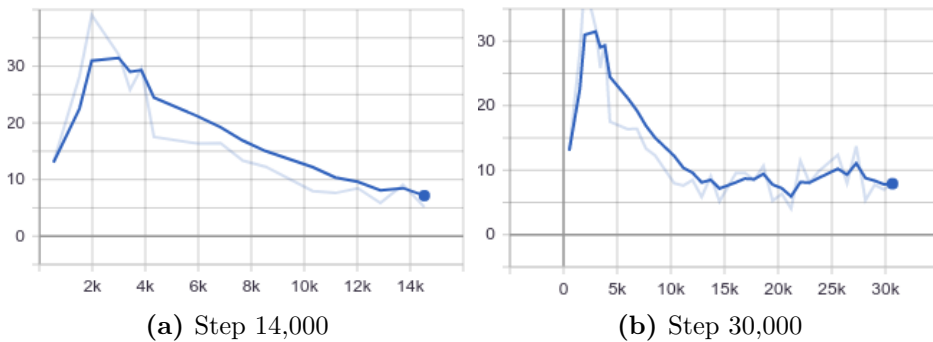
## Object Detection

This chapter evaluates the developed object detection model for the circular box shown in Figure 5.7. Section 10.1 presents results obtained for the model. These are evaluation metrics from the training process and probabilities obtained when experimenting with the model in the MANULAB. The ROS2 OpenVINO Toolkit and Object Analytics stack were applied for the experiments, together with the implemented ObjectDetection node. In Section 10.2, the results are evaluated, and it is discussed how the model and implementation could have been improved.

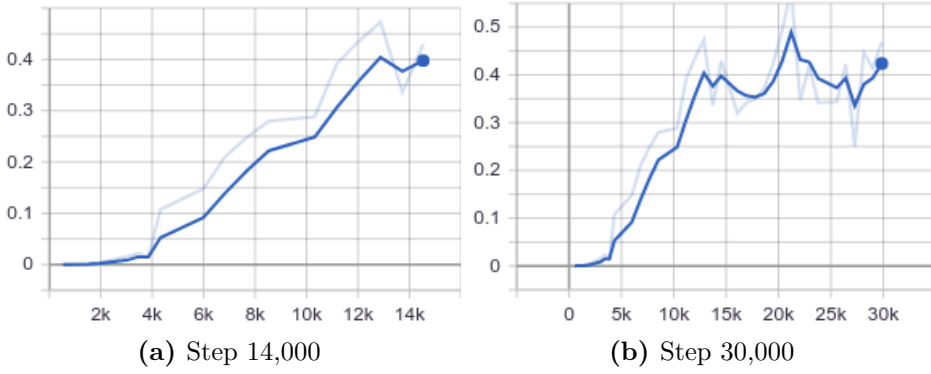
### 10.1. Results

#### 10.1.1. Training

Two common evaluation metrics in machine learning are loss and precision. It is desirable to have a low loss and a high precision. The metrics were monitored with TensorBoard during the training process. Figure 10.1 and 10.2 shows the evolution of the loss and precision through the training process.

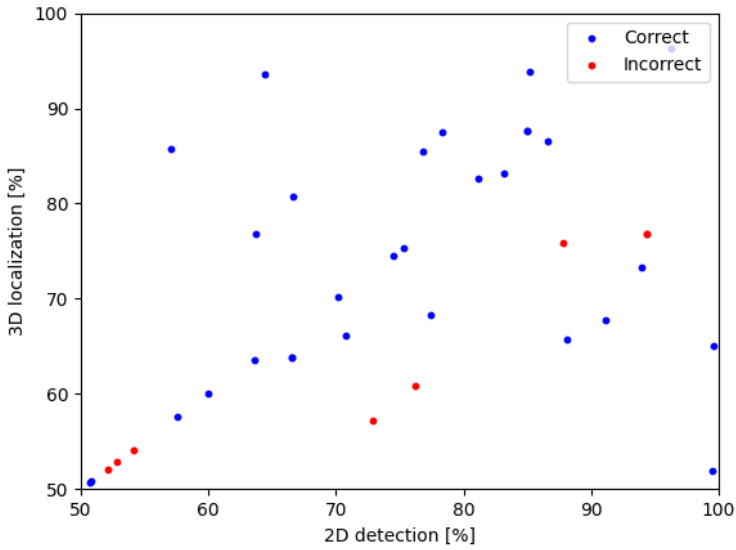


**Figure 10.1.:** Evolution of total loss during the training of the model.



**Figure 10.2.:** Evolution of precision during the training of the model.

### 10.1.2. Experiments



**Figure 10.3.:** Probabilities for 2D detection and 3D localization of the circular box during experiments with the model. The blue dots correspond to correctly recognized objects, while red dots represent incorrect classifications.

The same physical setup, as described in Section 9.1, is applied. The experiments are conducted by moving the manipulator between the defined search areas while searching. The poses of the detected objects are saved, and when the manipulator



arrives in the correct position, the pose of the last detected object is outputted. The probabilities for detected objects are shown in Figure 10.3. The blue dots correspond to correctly recognized objects, while red dots represent objects that are wrongly identified. A threshold of 50 % is specified for 2D detection. Hence, all of the probabilities are above this value.

The average probability for correctly detected objects is 73.8% for 2D detection and 75.5% for 3D localization. For the incorrect classifications, the average probabilities are 63.3% and 73.0%, respectively.

## 10.2. Discussion

### 10.2.1. Training

It is advisable to let the model reach a total loss value of at least 2, ideally 1 and lower, to achieve fair detection results [68]. In Figure 10.1, the loss reaches the value of 8. Hence, the model is not optimal, and a better model could have been obtained. As can be seen from the evolution, the graph stops decreasing around 14,000 steps and flattens. The training was stopped at step 30,661 as the total loss was not expected to decrease any further. As the evolution of the metrics flattened out, adding more steps to the training process would probably not improve the model. Considering the oscillating graph in Figure 10.2b, it can be seen that the model had reached its maximum precision value of 0.42 when the training was ended. The oscillations might indicate that the model is overfitted. However, experiments have shown that this does not seem to be the case. Other black objects were often classified as a circular box, independently of its shape. The training set mostly included images of the box but should have included similar objects for the model to be able to distinct the circular box. It is assumed that the quality of the data set is a possible source of error for the model. The images in the data set contain the object of interest in scenes with different lighting, background and scale. Still, the images were taken as snapshots from only seven different videos. To obtain a better model, more videos could have been used to provide more variation to the images. Better performance and quicker deployment could probably be achieved by applying a trained model from the Open Model Zoo. However, it was desired to develop a custom object detection model, and this was not considered an option. The detection system is thought to be a proof of concept and was deemed to be acceptable for the desired use.

### 10.2.2. Experiments

The probabilities visualized in Figure 10.3 supports the fact that the model is sufficient. It is observed that the probabilities for 2D detection and 3D localiza-

tion, to a great extent, correlate, which is expected. Even though, it was observed throughout the experiments that the model at times predicted wrong elements to be a circular box. The model rather recognized an object in the image, either incorrect or correct, than not detecting anything.

Another finding from the experiments was that the localization sometimes failed, even though the object was detected correctly. The bounding box output from the object detection algorithm was estimated too large. This caused a shift in the calculated center point, and the gripper did not approach the middle of the box. This was observed to occur when a shadow behind the box was included in the bounding box. The model was trained with images taken from different points of view, and from some angles, the box may seem oval due to the vanishing point phenomenon. When a shadow occurs behind a dark object, it looks like an extension of the object, and it may appear similar to as if the box were seen from a distance.

In the implemented `ObjectDetection` node, only the position of the center point for the 3D bounding box is calculated. Initially, the orientation was not set, corresponding to a pose aligned with the camera optical frame. This turned out to be incorrect as the trajectory for the manipulator is planned with respect to the frame of the center point of the gripper. A simple approach was to correct the orientation based on experimental findings manually. However, what should have been done was to calculate the orientation of the bounding box, based on its normals and boundaries.

It was desired to perform object detection only when the manipulator was in one of the search positions to reduce the processing of data. This could be performed by utilizing the pipeline service included in the OpenVINO Toolkit. The 2D detection outputs the objects found in the image frame in every message published. In contrast, the 3D localization only outputs the first occurrence of a bounding box. However, if the frame or the object is moving, each new bounding box is published. Because of this, it was not possible to start the object detection when the manipulator was in the search position and hence, not moving. An efficient solution would have been to start running the pipeline when the manipulator starts moving towards the search position. This could have been realized with parallel behavior tree nodes. As this behavior was discovered during the last period of experiments, a more straightforward solution was chosen. As of now, the detection pipeline is always running. The `ObjectDetection` node subscribes to the localization topic and saves the last observed instance of a 3D bounding box. If the object search is triggered, and the probability of the localization is sufficient, the pose of the previous occurrence is returned. As expected, this led to incorrect poses in scenarios where the last localization of the object was saved when the manipulator was far away from the search position.

## Chapter 11.

# Mobile Navigation and Manipulation

In this chapter, the components previously presented are composed to a comprehensive system. The composed robot system consists of the KMR, together with the additional cameras and gripper. The developed system implements functionality for the robot to perform a basic fetch and carry scenario. The experimental setup is presented in Section 11.1. Section 11.2 presents the results of the experiments and Section 11.3 evaluates the performance of the composed system.

### 11.1. Experimental Setup

The system was tested through two experiments. In the first experiment, the full system was launched. This includes the system described in the system description in addition to the addressed ROS stacks. In the second experiment, components related to navigation and KMP were omitted.

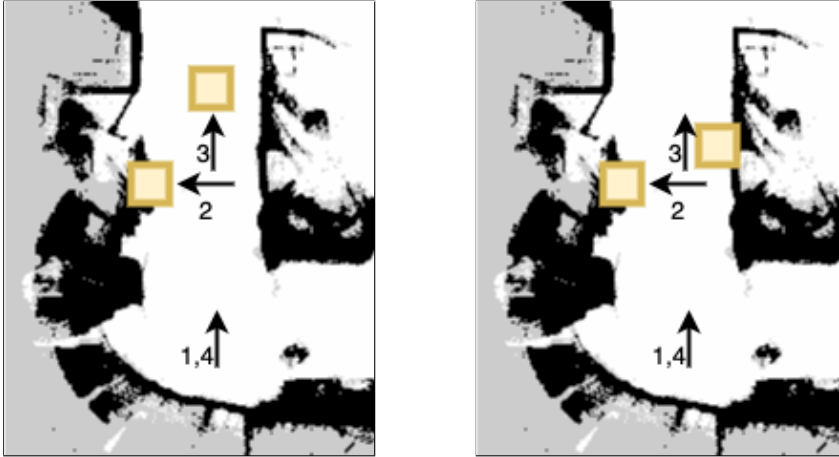
Figure 11.1 illustrates the experimental setup. Two workstations are set up by cardboard boxes with objects to be grasped and carried.

The behavior tree was executed to carry out the logic for the experiments, which can be simplified by the following sequence:

1. Navigate to goal
2. Move the arm to the defined search areas
3. Search for object
4. Move to object
5. Grasp object

6. Put the object in a carry area
7. Move the arm to the drive position

The listed sequence further includes subtasks and failure handling, as described in the system description. In the first experiment, two work stations were defined as goal poses, which means the total sequence was carried out two times. When the goal pose list is empty, the robot should move back to the start position. For the second experiment, task number 2 to 7 were carried out three times while the KMP was stationary at the first work station.



(a) Setup used in the video *composed1* (b) Setup used in the video *composed2*

**Figure 11.1.:** The experimental setup with defined work stations in the laboratory. The yellow boxes represent work stations, while the enumerated arrows defines the goal poses for the experiment.

## 11.2. Results

Three videos are provided as documentation of the experiments. They can be found in the digital attachments. The videos *composed1.mov* and *composed2.mov* show the first experiment with the setups in Figure 11.1a and 11.1b, respectively, while the video *manipulation.mov* shows the second experiment without navigation.

### 11.2.1. Video: *composed1*

The video shows that the robot drives between the two work stations and back to the initial position. The robot is not able to localize the object in the first work

station and moves on to the second. In the second work station, the object is recognized, but wrongly localized. This is a consequence of that the last occurrence of the object is stored simultaneously as the robot arm is moving. The robot tries to grasp the faulty localized object. However, as the gripper detects no object, the manipulator is moved back to the drive position, and the robot drives back to the initial position.

### 11.2.2. Video: *composed2*

The video shows that the robot drives between the two work stations and back to the initial position. In the first work station, the robot successfully localizes and grasp the object. In the second work station, the object is not recognized in the first search area, and the manipulator is moved to the second. When starting to move, the object is localized with respect to the first search area, and the pose is stored. When the manipulator reaches the second search area, a grasping motion is executed to the already stored pose as no other object was detected in this search position. The robot tries to grasp the faulty localized object, but no object is detected by the gripper, and the manipulator moves to the drive position. The video was ended while the robot navigated towards the initial position.

### 11.2.3. Video: *manipulation*

The video shows the KMR in front of the first work station, where two objects are located. The robot makes three attempts to pick up the two objects. The first attempt is successful, and the object is picked up and placed in the first carry area on the platform. In the second attempt, the position retrieved of the object is slightly incorrect, leading to that the gripper fails to grasp the object. As the gripper detects no object, the robot moves the arm back to the drive position. In the third attempt, the second object is successfully located and grasped. The object is then placed in the second carry area, as the first area is denoted as occupied in the blackboard of the behavior tree.

*Note: The power inverter used to supply power to the onboard computer did not work on the day the experiments were conducted. Due to this, an extension cord was used instead.*

## 11.3. Discussion

The behavior tree simplifies the process of linking components of the system together, creating a logical structure for sequences of tasks and failure handling. By defining different trees based on the implemented tree nodes, experiments for a

variety of scenarios can quickly be executed. This has been useful for testing and validation of the system, such as for the experiments conducted for the composed system in Section 11.2.

When developing software, an effective approach is to decompose the system and test each component individually. Hence, a prioritization was to ensure that the components of the system work as desired separately. As it has emerged through the results for each part of the system, this was a successful approach, as each component alone provides satisfactory results. The limited time in the laboratory affected the time available for testing the system as a whole. During the experiments conducted for the composed system, it was revealed that additional refinements should have been performed for executing the components sequentially. The points of improvement are minimal and are, to a large extent, concerned with the object localization part of the system. This is reflected in the videos *composed1* and *composed2*, where the navigation between the work stations is impeccable, but the localization of the object somehow fails. From the evaluation metrics of the object detection model, it was assumed poor performance of the model. This assumption was proven wrong by the result presented in video *manipulation*, where the robot is able to localize the object in 3 out of 3 attempts. The only error that occurs is due to the estimated position being slightly to the left of the center of the object. As discussed in Section 10.2, this might be due to the shadow of the object. Several attempts were conducted to carry out the fetch and carry scenario for the composed system, but the localization tended to fail. A common error was that the object was detected before the manipulator reached the search position, or that a similar object was detected in the background. This is, to a large extent, due to the implemented solution, which searches for objects simultaneously as the manipulator is moving. On the other hand, the same fault did not occur when the KMP was standing still. It is suspected that this is caused by latency in the network when several complex programs are executed at the same time and large amounts of data are transmitted. If the camera data input to the localization algorithm is delayed, this will lead to incorrect localization with respect to the pose of the moving manipulator.

Since the various components of the system might fail on their individual tasks, it is important to have a decision mechanism that performs fault handling in a proper manner, which the behavior trees manage. The failure handling of the behavior tree works as desired for the scenarios shown in the recordings. If the robot is not able to localize the box or pick it up, it simply moves on to its next task.

A further focus would have been to make the system work together and troubleshooting why the object detection component fails in the composed system. Overall, we are satisfied with the resulting system, considering the short time

available for testing.





Part III.

Conclusion



# Chapter 12.

## Conclusion

### 12.1. Further Work

The content presented as further work is divided into three categories: requirements, suggestions and going further. Requirements include adjustments which must be performed for further development of the system, while the second category present suggestions for improving the performance. The latter category describes interesting options for further potential of the system.

#### 12.1.1. Requirements

A challenge has been to interact Navigation2 with the safety restrictions for the monitored fields of the KMP. Due to the missing license and the restricted time in the laboratory, the final solution was to deactivate the emergency stop trigger for the protective fields. This is not a recommended solution and requires to be further investigated. A solution was proposed on how the monitored fields of the SICK laser scanners can be expanded. This way, one can react to observed obstacles before they enter the protective field, and the velocity can be reduced. In addition, the muting functionality will most likely be necessary to approach work stations and drive through narrow passages. The implemented NavigationSupport node makes it possible to dynamically adjust the velocities according to the status of the monitored field, such that interacting with the safety functionality should be an affordable process.

During the work, problems occurred when using the ROS Eloquent installation of Navigation2. This was due to the ROS C++ client library for ROS Eloquent. The necessary changes are fixed for the next ROS distribution, Foxy Fitzroy, which was released 5th of June 2020. The Foxy distribution also includes important upgrades for the Navigation2 stack. The distribution only provides support for

Ubuntu 20.04, which require that the operating systems of the external computers are upgraded.

As seen in the results for the composed system, refinements are required for the total system to work perfectly. This concerns improving the object detection model, the orientation for the detection and changing the logic relative to when searches are triggered. When this is conducted, further testing should be performed.

### 12.1.2. Suggestions

Currently, the two operating systems communicate utilizing TCP. UDP is the preferable option for real-time applications, and it is suggested to switch protocol eventually. Support for UDP is already implemented, and changing protocol only requires specific parameters to be changed.

During the time at the laboratory, there have been problems related to the network. If this problem persists, we recommend investigating this further. Proposed measures are to test whether improvements are observed with a UDP connection, investigate if there may be elements in the laboratory that interfere with the network and test with a different router.

The Java implementation has potential for improvement in the form of making the code more efficient and reduce processing. This includes utilizing more concepts from Java and RoboticsAPI, such as events, listeners and background applications. This may avoid the FDI connection from disconnecting in addition to a more continuous motion for the KMP.

Currently, the synchronization of clocks between the different computers of the system is not handled. The current solution is to overwrite the timestamps of the data received from Sunrise. The timestamps are set by the current time on the external computer before published to ROS. According to Thomas Rühr, this is a sufficient solution [1]. Alternatively, he suggests that the remote PC and the Navigation PC, outputting sensor data, can be synchronized with external [NTP](#).

The Robotiq controller is currently configured for Modbus-RTU, but can be re-configured to support EtherCAT. If this is done, the interface on the media flange can be utilized for integrating the mounted gripper to the Sunrise system. This would lead to a more elegant solution, instead of the controller being connected to the onboard computer as of today. As the 2F-85 gripper does not support EtherCAT directly, this solution would still require cables along the manipulator. The optimal solution would be to exchange the 2F-85 gripper with a tool that supports one of the interfaces at the media flange, such that unnecessary coupling is avoided. This enables integration with the Sunrise system, such that the weight of the tool can be considered during planning.

Regarding path planning for the manipulator, performed by MoveIt, multiple points of improvement were discussed in Section 9.3. The most important points are to test other planners, to avoid long, winding trajectories and include sensor data, and possibly an octomap, to enable collision checking with the environment.

### 12.1.3. Going Further

To fulfill the requirements for Industry 4.0 in an environment consisting of different types of robots and machines, information should be shared among the devices. An interesting use case for the industry is in a facility with several work stations where components are to be transferred for processing. The KMR needs to have insight to the processes carried out at the different stations, and commands should be given hereafter. Further, the mobile robot can be utilized to transport finished components to other robot cells and to assist processes.

The LBR, with its high precision and positioning, can perform exact machining and assembly tasks. Other tools than the gripper applied in our work can easily be mounted and integrated to expand the applications of the manipulator. An interesting scenario would be to cooperate with other robots in the laboratory. In this case, perception would be crucial for the robots not to collide and to carry out operations on specific components.

Two KMRs are present at the MANULAB. The system can be applied to operate both of the robots simultaneously. The implementation can be reused for the other robot, but functionality must be added for the two robots to interact with each other. If a component is ready to be transported from a work station, the closest of the two robots should be the one to carry out the task. Components that are larger than the maximum allowed load for an individual robot can be transported by distributing the weight onto both platforms. The two robots may also be programmed to cooperate on tasks that require two manipulators, such as assembling or lifting.

## 12.2. Concluding Remarks

The main objective of this thesis is to enable autonomous operation of the KMR iiwa in the MANULAB. The developed system utilizes the mobile base, the manipulator, the integrated sensors of the KMR, and the external sensors and actuators. An underlying objective for the work is to make the system applicable in the environment and with the tools available at the MANULAB. This involved testing different configurations and adapting the system to the resources.

A new architecture for the communication between the KMR and ROS2 was suc-

cessfully created, which satisfies the requirements of being more scalable, flexible and fault tolerant. The communication architecture has been verified through experiments with the various components of the system. An experimental platform has been proposed, with the necessary connections between the devices.

The addition of cameras to the mobile base was successful, both related to performing SLAM and improving the navigation. The final map, created with both laser scanners and cameras, includes all of the features in the environment and enables safer navigation in the environment.

Efforts have been made to improve the performance of Navigation2 when applied with the KMP. The main areas of focus have been to achieve smooth movements, dynamic replanning, obstacle avoidance and adapting the controller to the safety restrictions of the KMR. From previously just being compatible with the robot, the robot may now be navigated safely in a real environment.

To accomplish the objectives for manipulation, the communication architecture has been expanded, and control of the LBR has been enabled through compatibility with MoveIt2. Spline motions were found to be the best solution for accurate execution of the planned trajectories. Functionality for object localization and using a gripper is implemented, which makes it possible to pick up specific objects using the manipulator.

The use of behavior trees enabled components to be linked together. From experiments, it has been shown that the implemented behavior trees handle errors in a satisfactory manner when the individual parts of the system fail.

When executing a complete fetch and carry scenario between different work stations, the system did not perform as good as desired. The failure is, to a large extent, concerned with the object localization component. Referring to the attached movies for the scenario including navigation, the object was localized and correctly picked up in one out of four trials. Without navigation, the object was correctly localized three out of three times, and correctly picked up two out of three times. As the other parts of the system work as expected, the object detection component should be further developed.

Due to the Covid-19 situation, the MANULAB was closed for eight weeks, and after reopening, the access was restricted. The limited time in the laboratory influenced the time spent on testing the system as a whole, as it was prioritized to ensure that the components of the system work separately. The circumstances also affected priorities that were made along the way, as we depended on solutions that could be simulated and implemented without access to the robot. The system touches a wide specter of disciplines. It has been prioritized to create a comprehensive system, and specific parts of the system have been emphasized more. This applies to SLAM and navigation, as using the KMR safely was con-

sidered as crucial. The system has points of improvement, which are highlighted as further work. These are consequences of conscious choices made during the work or have emerged through experiments.

A comprehensive system has been achieved that can navigate autonomously around the environment and perform basic fetch and carry operations. It creates a foundation for being able to perform other interesting experiments with the KMR iiwa. The paper published by the authors emphasizes that this is a system with high relevance and potential. This is also supported by the enthusiasm of KUKA senior developer Thomas Ruhr after being introduced to our work. The final implemented solution has the desired functionality and performs acceptably.





# Bibliography

- [1] Thomas Rühr. *Mobile Manipulation with the KUKA KMR iiwa & ROS*. Dec. 2019. URL: [https://static1.squarespace.com/static/51df34b1e4b08840dcfd2841t/5e21b72ff52b107943e83dcf/1579267895764/03\\_Thomas\\_Ruehr\\_KUKA\\_ROS\\_I\\_2019\\_release\\_big.pdf](https://static1.squarespace.com/static/51df34b1e4b08840dcfd2841t/5e21b72ff52b107943e83dcf/1579267895764/03_Thomas_Ruehr_KUKA_ROS_I_2019_release_big.pdf) (visited on 05/31/2020).
- [2] Ricardo Tellez. *What is ROS?* Sept. 2019. URL: <https://www.theconstructsim.com/what-is-ros/> (visited on 06/09/2020).
- [3] Charlotte Heggem and Nina Marie Wahl. *Configuration and Control of KMR iiwa with ROS2*. Dec. 2019.
- [4] NTNU. *SIMS 2020 - 3rd International Symposium on Small-scale Intelligent Manufacturing Systems*. 2020. URL: <https://www.ntnu.edu/ieee-sims-2020/> (visited on 06/01/2020).
- [5] Andreas Dömel, Simon Kriegel, Michael Kaßecker, Manuel Brucker, Tim Bodenmüller, and Michael Suppa. *Toward fully autonomous mobile manipulation for industrial environments*. July 2017. DOI: [10.1177/1729881417718588](https://doi.org/10.1177/1729881417718588). (Visited on 05/16/2020).
- [6] Salvatore Virga and Marco Esposito. *IFL-CAMP/iiwa\_stack*. Apr. 2020. URL: [https://github.com/IFL-CAMP/iiwa\\_stack](https://github.com/IFL-CAMP/iiwa_stack) (visited on 04/28/2020).
- [7] Salvo Virga. *Do we have any plans to transplant iiwa to ROS2? · Issue #201 · IFL-CAMP/iiwa\_stack*. GitHub. Mar. 2019. URL: [https://github.com/IFL-CAMP/iiwa\\_stack/issues/201](https://github.com/IFL-CAMP/iiwa_stack/issues/201) (visited on 05/16/2020).
- [8] Thomas Rühr. *KMR iiwa and ROS*. E-mail. Feb. 2020.
- [9] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Texts in Computer Science. London: Springer London, 2011. ISBN: 978-1-84882-934-3 978-1-84882-935-0. DOI: [10.1007/978-1-84882-935-0](https://doi.org/10.1007/978-1-84882-935-0). URL: <http://link.springer.com/10.1007/978-1-84882-935-0> (visited on 03/26/2020).
- [10] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. 2nd ed. Cambridge, UK ; New York: Cambridge University Press, 2003. ISBN: 978-0-521-54051-3.
- [11] OpenCV. *Camera Calibration*. 2020. URL: [https://docs.opencv.org/master/dc/dbb/tutorial\\_py\\_calibration.html](https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html) (visited on 06/01/2020).

- [12] Noah Snavely. *Lecture 23: Structure from motion and multi-view stereo*. 2015. URL: <https://slideplayer.com/slide/4952501/>.
- [13] C. Harris and M. Stephens. “A Combined Corner and Edge Detector”. In: *Proceedings of the Alvey Vision Conference 1988*. Alvey Vision Conference 1988. Manchester: Alvey Vision Club, 1988. DOI: [10.5244/C.2.23](https://doi.org/10.5244/C.2.23). URL: <http://www.bmva.org/bmvc/1988/avc-88-023.html> (visited on 05/16/2020).
- [14] Himanshu Vaghela, Manan Oza, and Sudhir Bagul. *Morphological Retina Keypoint Descriptor*. Jan. 2019. URL: <http://arxiv.org/abs/1901.08213> (visited on 06/01/2020).
- [15] Mehmet Hacibeyoglu. (PDF) *Human Gender Prediction on Facial Mobil Images using Convolutional Neural Networks*. 2018. URL: [https://www.researchgate.net/publication/328405250\\_Human\\_Gender\\_Prediction\\_on\\_Facial\\_Mobil\\_Images\\_using\\_Convolutional\\_Neural\\_Networks/figures?lo=1](https://www.researchgate.net/publication/328405250_Human_Gender_Prediction_on_Facial_Mobil_Images_using_Convolutional_Neural_Networks/figures?lo=1) (visited on 06/08/2020).
- [16] Alexander J B Trevor, Suat Gedikli, Radu B Rusu, and Henrik I Christensen. *Efficient Organized Point Cloud Segmentation with Connected Components*. 2013. URL: [https://cs.gmu.edu/~kosecka/ICRA2013/spme13\\_trevor.pdf](https://cs.gmu.edu/~kosecka/ICRA2013/spme13_trevor.pdf).
- [17] Carol Fairchild and Dr Thomas L. Harman. *ROS Robotics By Example: Learning to control wheeled, limbed, and flying robots using ROS Kinetic Kame*. Packt Publishing Ltd, Nov. 2017. ISBN: 978-1-78847-472-6.
- [18] Cyrill Stachniss. *Introduction to Robot Mapping*. 2012. URL: <http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam01-intro-4.pdf>.
- [19] Doaa M. A. Latif, Mohammed A. Megeed Salem, H. Ramadan, and Mohamed I. Roushdy. *3D Graph-based Vision-SLAM Registration and Optimization*. 2014. URL: <http://www.naun.org/main/NAUN/circuitssystemssignal/2014/a162005-171.pdf> (visited on 04/24/2020).
- [20] Steve Macenski. *ROSCon 2019 Macau: On Use of SLAM Toolbox*. Dec. 2019. URL: <https://vimeo.com/378682207> (visited on 04/24/2020).
- [21] Trym Vegard Haavardsholm. *A handbook in Visual SLAM*. Sept. 24, 2019.
- [22] Mathieu Labbé and François Michaud. *RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation*. Mar. 2019. DOI: [10.1002/rob.21831](https://doi.org/10.1002/rob.21831). (Visited on 05/09/2020).
- [23] KUKA Deutschland GmbH. *Mobile Robots KMR iiwa omniMove Mobile Industrial Robot System Assembly and Operating Instructions*. Vol. PB9761. 2019. URL: [xpert.kuka.com](http://xpert.kuka.com).

- [24] KUKA AG. *KMR iiwa - Mobile robot by KUKA AG / DirectIndustry*. 2016. URL: <https://www.directindustry.com/prod/kuka-ag/product-17587-1714901.html> (visited on 06/09/2020).
- [25] KUKA Deutschland GmbH. *System Software KUKA Sunrise.OS 1.16 KUKA Sunrise.Workbench 1.16 Operating and Programming Instructions for System Integrators*. Vol. PB10796. Dec. 2019. URL: [xpert.kuka.com](http://xpert.kuka.com).
- [26] Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Michael Vistein, and Wolfgang Reif. *The Robotics API: An object-oriented framework for modeling industrial robotics applications*. Taipei, Oct. 2010. DOI: [10.1109/IROS.2010.5649098](https://doi.org/10.1109/IROS.2010.5649098). (Visited on 05/31/2020).
- [27] SICK AG. *Safety laser scanners / S300 Standard SICK*. 2019. URL: <https://www.sick.com/ag/en/opto-electronic-protective-devices/safety-laser-scanners/s300-standard/c/g187239> (visited on 06/08/2020).
- [28] SICK AG. *Operating Instructions S300 Safety Laser Scanner SICK Sensor Intelligence*. Vol. 8010948/YY96/2016-02-17. Feb. 2016. URL: [https://cdn.sick.com/media/docs/3/13/613/Operating\\_instructions\\_S300\\_Safety\\_laser\\_scanner\\_en\\_IM0017613.PDF](https://cdn.sick.com/media/docs/3/13/613/Operating_instructions_S300_Safety_laser_scanner_en_IM0017613.PDF).
- [29] KUKA Roboter GmbH. *Mobile Robots KMP 200 omniMove Transport Vehicle Operating Instructions*. Vol. PB9761. 2017. URL: [xpert.kuka.com](http://xpert.kuka.com).
- [30] KUKA Deutschland GmbH. *KUKA Software Option KUKA Sunrise.Mobility 1.11 For KUKA Sunrise.OS 1.16 For KUKA Sunrise.Workbench 1.16*. Vol. PB13250. Aug. 2019. URL: [xpert.kuka.com](http://xpert.kuka.com).
- [31] KUKA Deutschland GmbH. *System Software KUKA.NavigationSolution 1.14 Operating and Programming Instructions for System Integrators*. Vol. PB13076. June 2019. URL: [xpert.kuka.com](http://xpert.kuka.com).
- [32] Wikipedia. *PROFINET*. Wikipedia. 2005. URL: <https://en.wikipedia.org/wiki/PROFINET> (visited on 11/02/2019).
- [33] KUKA Deutschland GmbH. *Robots LBR iiwa LBR iiwa 7 R800, LBR iiwa 14 R820 Specification*. Vol. PB2535. KUKA, May 2019. URL: [xpert.kuka.com](http://xpert.kuka.com).
- [34] robots.ieee.org/. *LBR iiwa - ROBOTS: Your Guide to the World of Robotics*. 2013. URL: <https://robots.ieee.org/robots/lbriiwa/> (visited on 03/23/2020).
- [35] KUKA Roboter GmbH. *Robot Option Media Flange For Product Family LBR iiwa Assembly and Operating Instructions*. Vol. Option Media Flange V8. Aug. 2016. URL: [xpert.kuka.com](http://xpert.kuka.com) (visited on 03/23/2020).

- [36] Robotiq.com and leanrobotics.org. *Robotiq 2F-85 & 2F-140 for CB-Series Universal Robots*. July 2018. URL: [https://assets.robotiq.com/website-assets/support\\_documents/document/2F-85\\_2F-140\\_Instruction\\_Manual\\_CB-Series\\_PDF\\_20190329.pdf](https://assets.robotiq.com/website-assets/support_documents/document/2F-85_2F-140_Instruction_Manual_CB-Series_PDF_20190329.pdf) (visited on 03/23/2020).
- [37] Modbus.org. *MODBUS over Serial Line Specification and Implementation Guide V1.02*. Dec. 2006. URL: [http://www.modbus.org/docs/Modbus\\_over\\_serial\\_line\\_V1\\_02.pdf](http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf) (visited on 03/23/2020).
- [38] Intel Realsense Technology. *Depth Camera D435*. Intel® RealSense™ Depth and Tracking Cameras. 2019. URL: <https://www.intelrealsense.com/depth-camera-d435/> (visited on 03/23/2020).
- [39] Intel Realsense Technology. *Intel RealSense D400 Series Product Family Datasheet*. Jan. 2019. URL: <https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/Intel-RealSense-D400-Series-Datasheet.pdf> (visited on 03/23/2020).
- [40] Intel Corporation. *ROS2 Wrapper for Intel® RealSense™ Devices*. 2019. URL: [https://github.com/intel/ros2\\_intel\\_realsense](https://github.com/intel/ros2_intel_realsense) (visited on 03/23/2020).
- [41] Intel Realsense Technology. *Intel RealSense D400 Series Calibration Tools - User Guide*. Jan. 2019. URL: [https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense\\_D400\\_Dyn\\_Calib\\_User\\_Guide.pdf](https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense_D400_Dyn_Calib_User_Guide.pdf) (visited on 03/23/2020).
- [42] Open Source Robotics Foundation. *ROS on DDS*. 2014. URL: [http://design.ros2.org/articles/ros\\_on\\_dds.html](http://design.ros2.org/articles/ros_on_dds.html) (visited on 04/28/2020).
- [43] Open Source Robotics Foundation. *ROS 2 and different DDS/RTPS vendors*. 2018. URL: <https://index.ros.org/doc/ros2/Concepts/DDS-and-ROS-middleware-implementations/> (visited on 04/28/2020).
- [44] Open Source Robotics Foundation. *About ROS2 client libraries*. 2019. URL: <https://index.ros.org/doc/ros2/Concepts/ROS-2-Client-Libraries/> (visited on 04/28/2020).
- [45] Open Source Robotics Foundation, Inc. *ROS2 Actions*. 2019. URL: <http://design.ros2.org/articles/actions.html> (visited on 04/27/2020).
- [46] The Cartographer Authors. *Algorithm walkthrough for tuning — Cartographer ROS documentation*. 2018. URL: [https://google-cartographer-ros.readthedocs.io/en/latest/algo\\_walkthrough.html](https://google-cartographer-ros.readthedocs.io/en/latest/algo_walkthrough.html) (visited on 05/23/2020).
- [47] Sameer Agarwal and Keir Mierle. *Ceres Solver*. 2018. URL: <http://ceres-solver.org>.

- [48] Introlab. *RTAB-Map*. RTAB-Map. 2019. URL: <http://introlab.github.io/rtabmap/> (visited on 05/10/2020).
- [49] Davide Faconti. *BehaviorTree.CPP*. 2019. URL: <https://www.behaviortree.dev/> (visited on 04/20/2020).
- [50] ros-planning. *ROS2 Navigation2 documentation*. 2020. URL: <https://navigation.ros.org/> (visited on 06/01/2020).
- [51] Yoonseok Pyo, Hancheol Cho, Leon Jung, and Darby Lim. *ROS Robot Programming (English)*. ROBOTIS, Dec. 2017. ISBN: 979-11-962307-1-5.
- [52] PickNik Robotics. *MoveIt Motion Planning Framework*. 2020. URL: <https://moveit.ros.org/> (visited on 03/24/2020).
- [53] PickNik Robotics. *MoveIt 2 Beta Release Announcement | MoveIt*. 2020. URL: <https://moveit.ros.org/moveit/ros2/2020/02/18/moveit-2-beta-feature-list.html> (visited on 03/24/2020).
- [54] PickNik Consulting. *Planners | MoveIt*. URL: <https://moveit.ros.org/documentation/planners/> (visited on 04/29/2020).
- [55] Rice Kavraki Lab. *The Open Motion Planning Library*. 2020. URL: <http://ompl.kavrakilab.org/index.html> (visited on 03/26/2020).
- [56] PickNik Robotics. *Concepts, MoveIt*. 2020. URL: <https://moveit.ros.org/documentation/concepts/> (visited on 03/26/2020).
- [57] GAMMA research group. *FCL: A Flexible Collision Library*. 2013. URL: [http://gamma.cs.unc.edu/FCL/fcl\\_docs/webpage/generated/index.html](http://gamma.cs.unc.edu/FCL/fcl_docs/webpage/generated/index.html) (visited on 03/26/2020).
- [58] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. *OctoMap: an efficient probabilistic 3D mapping framework based on octrees*. Apr. 2013. DOI: [10.1007/s10514-012-9321-0](https://doi.org/10.1007/s10514-012-9321-0). (Visited on 03/26/2020).
- [59] Orocos. *Orocos Kinematics and Dynamics C++ library*. Mar. 2020. URL: [https://github.com/orocos/orocos\\_kinematics\\_dynamics](https://github.com/orocos/orocos_kinematics_dynamics) (visited on 03/26/2020).
- [60] Intel Corporation. *Intel Robot DevKit*. 2019. URL: [https://github.com/intel/robot\\_devkit](https://github.com/intel/robot_devkit).
- [61] Intel Corporation. *openvinotoolkit/openvino*. GitHub. 2020. URL: <https://github.com/openvinotoolkit/openvino> (visited on 05/01/2020).
- [62] Intel Corporation. *Model Optimizer Developer Guide - OpenVINO™ Toolkit*. 2020. URL: [https://docs.openvinotoolkit.org/latest/\\_docs\\_MO\\_DG\\_Deep\\_Learning\\_Model\\_Optimizer\\_DevGuide.html](https://docs.openvinotoolkit.org/latest/_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html) (visited on 04/01/2020).

- [63] Intel Corporation. *intel/ros2\_object\_analytics*. 2019. URL: [https://github.com/intel/ros2\\_object\\_analytics](https://github.com/intel/ros2_object_analytics) (visited on 03/31/2020).
- [64] Tzutalin. *LabelImg*. 2015. URL: <https://github.com/tzutalin/labelImg> (visited on 03/31/2020).
- [65] Google Brain Team. *TensorFlow*. URL: <https://www.tensorflow.org/?hl=nb> (visited on 03/31/2020).
- [66] Jonathan Huang, Vivek Rathod, Ronny Votel, Chen Sun, Menglong Zhu, Alireza Fathi, and Zhichao Lu. *TensorFlow Object Detection API*. 2020. URL: <https://github.com/tensorflow/models> (visited on 03/31/2020).
- [67] Google Brain Team. *Tensorflow detection model zoo*. 2019. URL: [https://github.com/tensorflow/models/blob/master/research/object\\_detection/samples/configs/ssd\\_inception\\_v2\\_coco.config](https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_inception_v2_coco.config) (visited on 03/31/2020).
- [68] Lyudmil Vladimirov. *Training Custom Object Detector — TensorFlow Object Detection API tutorial documentation*. 2018. URL: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html> (visited on 03/31/2020).
- [69] Intel Corporation. *ROS1 Wrapper for Intel® RealSense™ Devices*. Apr. 2020. URL: <https://github.com/IntelRealSense/realsense-ros> (visited on 04/23/2020).
- [70] Daniel Ordonez. *Robotiq 2finger grippers*. Apr. 2020. URL: [https://github.com/Danfoa/robotiq\\_2finger\\_grippers](https://github.com/Danfoa/robotiq_2finger_grippers) (visited on 04/23/2020).
- [71] ros-planning. *ros-planning/navigation2*. Apr. 2020. URL: <https://github.com/ros-planning/navigation2> (visited on 04/20/2020).
- [72] Oracle Corporation. *Multithreading Concepts*. 2010. URL: <https://docs.oracle.com/cd/E19253-01/816-5137/mtintro-25092/index.html> (visited on 05/26/2020).
- [73] Nicolas Lauzier. *The Right Gripper for the Kuka Lightweight Robot (LWR)*. [blog.robotiq.com/](http://blog.robotiq.com/). May 2016. URL: <https://blog.robotiq.com/bid/56804/The-Right-Gripper-for-the-Kuka-Lightweight-Robot-LWR> (visited on 03/23/2020).
- [74] ZoBotics. *ELZo3/RobotiqGripper-2F85-EtherCat-Control*. July 2019. URL: <https://github.com/ELZo3/RobotiqGripper-2F85-EtherCat-Control> (visited on 03/23/2020).
- [75] ros-perception. *Depthimage\_to\_laserscan*. 2019. URL: [https://github.com/ros-perception/depthimage\\_to\\_laserscan/tree/ros2](https://github.com/ros-perception/depthimage_to_laserscan/tree/ros2).
- [76] ros-perception. *pointcloud\_to\_laserscan*. 2019. URL: [https://github.com/ros-perception/pointcloud\\_to\\_laserscan](https://github.com/ros-perception/pointcloud_to_laserscan).

- [77] Steve Macenski. *SteveMacenski/slam\_toolbox*. Apr. 2020. URL: [https://github.com/SteveMacenski/slam\\_toolbox](https://github.com/SteveMacenski/slam_toolbox) (visited on 04/24/2020).
- [78] Mathieu Labbé. *Kinect mapping*. GitHub. June 2018. URL: <https://github.com/introlab/rtabmap> (visited on 05/24/2020).
- [79] Kevin M. Lynch and Frank C. Park. *Modern robotics: mechanics, planning, and control*. Cambridge, UK: Cambridge University Press, 2017. ISBN: 978-1-107-15630-2 978-1-316-60984-2.
- [80] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. *Real-time loop closure in 2D LIDAR SLAM*. Stockholm, Sweden, May 2016. DOI: [10.1109/ICRA.2016.7487258](https://doi.org/10.1109/ICRA.2016.7487258). (Visited on 05/10/2020).
- [81] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1-4493-2389-8.
- [82] Steve Macenski. *Replace Navfn with Hybrid approach · Issue #1279*. GitHub. Oct. 2019. URL: <https://github.com/ros-planning/navigation2/issues/1279> (visited on 05/30/2020).
- [83] Brian Wilcox. *DWB: PathAlign and GoalAlign critics lead to poor obstacle avoidance #938*. GitHub. 2019. URL: <https://github.com/ros-planning/navigation2/issues/938> (visited on 06/05/2020).
- [84] ros-planning. *OMPL Planner*. Jan. 2020. URL: [https://ros-planning.github.io/moveit\\_tutorials/doc/ompl\\_interface/ompl\\_interface\\_tutorial.html](https://ros-planning.github.io/moveit_tutorials/doc/ompl_interface/ompl_interface_tutorial.html) (visited on 06/02/2020).





# Appendix



# Appendix A.

## Digital Attachments

Four videos are provided as digital attachments to the thesis as documentation of conducted experiments:

- Video: *replanning.mp4*
- Video: *composed1.mov*
- Video: *composed2.mov*
- Video: *manipulation.mov*

Video *replanning* is presented in the results for dynamic replanning Section [8.1.2](#). Video *composed1*, *composed2* and *manipulation* are presented in the results for the comprehensive system in Section [11.2](#).

The specialization project report is provided as an attachment to the thesis as it is fundamental for the conducted work:

- PDF: *Configuration and Control of KMR iiwa with ROS2*



## Appendix B.

# Conference Paper

The authors published a paper in context of the 3rd International Symposium on Small-scale Intelligent Manufacturing Systems (SIMS 2020). The conference is arranged at NTNU Gjøvik from the 10th to the 12th of June 2020. The paper presents the work conducted from August 2020 to March 2020. The authors will present their work as a part of the conference on the 11th of June.

# Configuration and Control of KMR iiwa Mobile Robots using ROS2

Charlotte Heggem\*, Nina Marie Wahl\* and Lars Tingelstad  
 Department of Mechanical and Industrial Engineering  
 NTNU, Norwegian University of Science and Technology  
 Trondheim, Norway

**Abstract**—In this paper, we present a novel system for controlling a KMR iiwa mobile robot using ROS2. The KMR iiwa is a mobile robot with a manipulator mounted on the base that is developed by the robot manufacturer KUKA. The developed system integrates with the Sunrise.OS operating system of the mobile robot and exposes sensor and control interfaces over UDP and TCP sockets. The controllability of the mobile robot from ROS2 is verified using the Cartographer and Navigation2 projects.

**Index Terms**—KMR iiwa, ROS2, KUKA, Mobile Robot, Industry 4.0

## I. INTRODUCTION

Industry 4.0 is about digitalization, automation, machine learning, and real-time data. A company that works extensively with industry 4.0 is the German company KUKA, Keller und Knappich Augsburg. The company produces robot systems for the industry and automated production solutions, and focus on networked, intelligent production. Such a production robot is the KMR iiwa, KUKA Mobile Robot Intelligent Industrial Work Assistant, shown in Figure 1.

The KMR iiwa is composed of a robot arm, the LBR iiwa 14 R820, and a mobile platform, the KMP 200 omniMove [1]. LBR is a German abbreviation for lightweight robot, while KMP is short for KUKA Mobile Platform. The intended use of the mobile robot is to handle automated manufacturing tasks and transport components, and it is characterized by its high degree of mobility and flexibility.

The operating system for the KMR iiwa mobile robots is KUKA Sunrise.OS. Programming the software requires knowledge of the Java programming language and the Sunrise software, including robot specific functions. This could be a blockade for many developers. Another drawback is that data and information from the robot are only available locally in the Sunrise system during executions of applications. In a decade when everything should be online, compatibility between networked devices is desired, and information should be available. Tools that support system integration based on common platforms are essential for the continuous evolution of Industry 4.0.

This work was supported by the Norwegian Research Council project MANULAB: Norwegian Manufacturing Research Laboratory under grant 269898.

\*Author names in alphabetical order. Charlotte Heggem and Nina M. Wahl contributed equally to this work.

978-1-7281-6419-9/20/\$31.00 ©2020 IEEE



Fig. 1. The KMR iiwa is composed of a mobile platform and a lightweight manipulator [2]. Image courtesy of KUKA Nordic AB.

ROS2, Robot Operating System 2, is the second generation of the open source framework ROS for developing robot applications with support for several programming languages and platforms [3]. Integration between Sunrise.OS and ROS2 is desirable as it would limit the required preknowledge required to operate the robot and make it more available for users. The ROS2 packages Cartographer and Navigation2 perform real-time localization, mapping and navigation of mobile vehicles and are highly relevant to use with the KMR iiwa.

Two projects with focus on interaction between the LBR iiwa and ROS have been inspirational for the work presented in this paper. The LBR iiwa and the KMP is operated by the same controller, and hence a similar approach for programming the robot can be applied to the KMP.

Virga and Esposito [4], propose an architecture with native ROSJava nodes launched on the robot controller. Hence, the two operating systems can exchange data and commands in the form of ROS messages over the ROS framework. The proposed solution requires installation of third-party libraries on the robot controller to run ROS nodes.

The work by Mokaram et al. [5] provides a simple standalone application that requires minimal installation on the robot controller. The architecture of the API consists of two main components, a single Sunrise.OS application on the robot controller and a ROS node launched on the external computer. The ROS node establishes a connection to the controller over TCP, Transmission Control Protocol. Data and command

messages are transmitted between the two components as strings.

The two APIs of [4] and [5] are developed for different purposes, but the implementations contain the same fundamental functionality. The main difference between the two architectures is whether to publish ROS messages directly from the robot controller or to utilize a middle-layer to handle the communication. The API of [5] includes simple methods for directly controlling the robot, while [4] present a more advanced system, including functionality for simulation in Gazebo and using the robot with ROS packages as MoveIt!

Work related to integration of KUKA mobile platforms and ROS has previously been explored to a small extent. Dömel et. al. present a concept toward fully autonomous mobile manipulation using ROS and KUKA omniRob, built on a different control system than the KMR iiwa [6]. However, the project is not open source.

In this paper, we present a communication interface between the KMR iiwa and ROS2 to control the mobile platform. The developed system integrates with the Sunrise.OS operating system of the mobile robot and exposes sensor and control interfaces over UDP, User Datagram Protocol, and TCP sockets. The interface is available online [7], and is the first free and open approach to controlling a KMR iiwa using ROS2.

The main functionality is autonomous navigation in an unknown, dynamic environment without collision. The goal is that a user should easily be able to connect to the robot and control it without any preknowledge about KUKA robotic systems.

The paper is structured as follows. A description of the KMR iiwa is presented in Section II. Further, in Section III, an introduction to ROS2 and the packages Cartographer and Navigation2 is given. A challenge with the implementation was to find the correct methods to command and retrieve data from the robot. The approach for obtaining the correct methods is described in Section IV. Next, in Section V, follows a system description with the implemented functionality of the interface. Section VI presents a verification of the functionality of developed system. Finally, Section VII concludes the paper.

## II. KMR IIWA

### A. Hardware

The KMP mobile platform has four mecanum wheels, which allow the mobile platform to move omnidirectionally. It is equipped with two SICK S300 safety laser scanners mounted diagonally opposite of each other.

The S300 is a compact laser measurement system that scans the environment in two dimensions in the height of 150 mm above the ground in means of infrared laser beams. Each laser has a scanning range of 270°, covering one long side and one short side of the vehicle, and a resolution of 0.5°.

The controller of the robot system, Sunrise Cabinet, is contained inside the KMP. The Sunrise Cabinet contains two PCs: one control PC and one navigation PC.

### B. Software

KUKA Sunrise.OS is the system software for robots that are operated by the Sunrise Cabinet. It offers functionality for programming and configuration of robot applications. Commands, sensor data, and information related to the ongoing operation are only available locally on the robot control system, or the supplied teach pendant, the SmartPAD.

### C. Operation

There are three different approaches for controlling the KMP: manually, autonomously, or by an application.

The most interesting option in this context is to control the KMP by a Java-based Sunrise application on the Sunrise Cabinet. The application can be implemented in the programming environment Sunrise.Workbench. The software packages KUKA RoboticsAPI and KUKA Sunrise.Mobility contain methods for obtaining information from the robot system and executing motions, which are the basis for developing a program for controlling the KMP.

KUKA.NavigationSolution is an optional software package with functionality for autonomous navigation of mobile platforms. The navigation software is based on sensor data from the S300 laser scanners and the odometry of the mobile platform.

The KMP can be moved manually by jogging the robot from the SmartPAD or the Radio Control Unit, an optional device with joysticks for controlling the platform.

The robot system has three different operation modes. T1 and T2 are manual operation modes used for testing and verification of programs. AUT is autonomous mode, which is the operating mode for program execution.

### D. Safety

The primary function of the S300 sensors is to operate as the safety equipment of the system by monitoring predefined areas around the vehicle. By default, the S300 sensors are configured with a protective field and a warning field. The size of the monitored fields depends on the velocity of the vehicle. The consequence of a violation of the two fields varies with the operation mode. Generally, a breach of a field causes a reduction of maximum travel speed or triggers a safety stop of the vehicle. The laser scanners are not active for velocities below 0.13 m/s in the manual operation modes.

## III. ROS2

ROS [3] is a robot operating system that can be used with multiple programming languages and has implemented open source functionality. It includes tools and libraries to handle the programming of robots without having to deal with hardware. The main goal of ROS is to provide a standard that can be used by any robot. ROS2, the second generation of ROS, is state-of-the-art software and is currently under massive deployment.

In general, ROS2 is cleaner and faster than the prior version, in addition to more flexible and universal. One of the main differences between the two versions is that ROS2 is built

on top of DDS, Data Distribution Service, which provides a distributed discovery feature.

Two ROS2 packages that are relevant for controlling the KMP is Cartographer [8] and Navigation2 [9].

Cartographer is a package for real-time SLAM, simultaneous localization and mapping, and is part of Google's open source projects [10].

A map can be created based on the robot's odometry, transformation information, and sensor information when the robot moves. The map can further be provided to Navigation2 and be used for navigation in the environment. The requirement for the cartographer node is sensor data measuring the distance to obstacles in the environment. Data from IMU sensors and odometry sensors can be included to improve the result.

Navigation2 is a package that can be used to control mobile robots and is based on a velocity controller. The main goal of applying the package is to navigate the robot from a start pose to a goal pose. This task can be broken down into subtasks like handling maps, localization of the robot, obstacle avoidance, and path following. When an obstacle-free path is calculated, velocity commands are produced to describe how the robot should move to follow the path. The navigation package requires information about the environment and how the robot moves, which can be provided in the form of a map, sensor data, and odometry data.

#### IV. APPROACH

The information required by Cartographer and Navigation2 defines the functional requirements of the system: It must be possible to retrieve odometry from the wheel encoders and laser data from the SICK scanners on the KMP, and the vehicle must be able to be controlled by velocity messages.

The motion commands and sensor retrieval methods from KUKA RoboticsAPI are limited and are chosen based on KUKA's definition of what is the intended use for programming the robot. A challenge that was faced during the development was to find the appropriate methods to retrieve data and move the KMP for the desired outcome.

As mentioned in Section II-D, the main functionality of the laser scanners is to monitor predefined areas around the vehicle. Boolean signals from the lasers indicate whether the monitored fields are violated, and can be extracted through defined methods from the KUKA RoboticsAPI. Range data from the laser scanners are only available through KUKA.NavigationSolution, and there are no direct methods to retrieve sensor data through KUKA RoboticsAPI.

The correct method for retrieving the sensor data was found by investigating the underlying functionality of KUKA.NavigationSolution. This software package introduces several views that can be opened in Sunrise Workbench, among them are the LaserView that visualizes range readings in real-time. By exploring the source code of the view, it was found that a FDI, Fast Data Interface, connection is established to the Navigation PC. The FDI connection utilizes a UDP socket to transmit data and enables functionality for subscribing to sensor data. As KUKA.NavigationSolution

relies on the odometry data from the KMP for navigation, this data can also be subscribed to through the FDI connection.

As for the motion commands, the predefined methods available from KUKA RoboticsAPI provide functionality for moving the KMP to an absolute or relative pose. This is not applicable for the intended use, as Navigation2 sends velocity commands.

The approach leading to the proposed solution was to investigate the underlying code of the devices used to jog the KMP manually. When a jogging button is held on the smartPAD or the joysticks are used to move the robot with the Radio Control Unit, the robot is continuously jogged. In a similar manner, a jog method can be executed continuously from a Java program to make the robot move with the specified velocity. The method found to jog the robot enabled access to the internal functionality of the robot, which is mainly marked as private and not intended to use for programming by external users.

By investigation of the source code, it was found that for each time the jog method is executed, a new thread is established. As the method is intended for internal use, the threads are marked as private, and there is no way to handle all the threads established when continuously calling the method. Over time this lead to an accumulation of threads, which is a problem when the number of threads get too high. A solution to this problem was found in the code of the Radio Control Unit, which revealed that a support class had to be implemented to handle the threads. The support class creates a thread pool for each new jogging execution, which can be shut down when the velocity motion is finished, and hence, kill all the threads established.

With the jogging motion it is possible to execute motion based on velocity commands, which further makes it possible to move the KMP by the commands from Navigation2.

#### V. SYSTEM DESCRIPTION

The implemented interface has the following main functionalities:

- Retrieve laser data from the KMP
- Retrieve odometry data from the KMP
- Retrieve status information from the KMP
- Move the KMP by giving velocity commands in the terminal
- Move the KMP by setting a goal pose in the terminal
- Use Cartographer to create a map of the environment
- Use Navigation2 to move the KMP

The interface consists of multiple ROS2 nodes running on a remote PC communicating with a Java program over TCP. The remote PC does not need to be in the immediate vicinity of the KMR, but must be connected to the same network. The Java program, *KMRiwaSunriseApplication*, is installed on the Control PC in the Sunrise Cabinet, which handles further communication with other internal devices on the robot. The physical architecture is shown in Figure 2.

Figure 3 shows the architecture of the implemented communication interface between the Java application and the



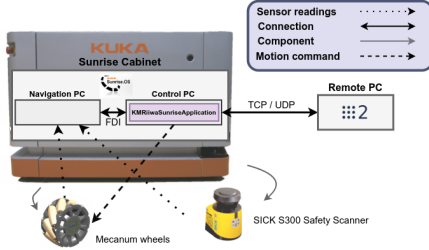


Fig. 2. Graphical representation of the physical system. Component images courtesy of KUKA Global.

ROS2 nodes. Each node is responsible for one area and is communicating with a corresponding Java class on the Sunrise Cabinet over UDP or TCP. Both protocols are implemented, and the desired communication type can be set in the system parameters. By creating separate nodes and connections for all tasks, it is possible to launch only the nodes needed for a specific use case. This limits the transferred data and enables a faster system.

The information is transmitted with the selected protocol as strings on the following format:

$$\underbrace{Length}_{\text{Additional Information}} > \underbrace{Type \ (LaserID) \ Timestamp \ Data}_{\text{Message}}$$

The additional information, *Length*, is only included if the message is sent over TCP. As TCP is a buffer protocol, this is necessary to ensure that the whole message is read.

The *Type* field describes what kind of data this message includes, and is necessary to know how the data should be processed. If the message is coming from the robot, an example of a type could be *Odometry*. An odometry message contains data describing the pose and velocity of the robot.

The pose of the robot is represented by a position vector  $x \in \mathbb{R}^3$  and a unit quaternion  $q \in S^3$ , while the velocity is represented by a twist  $\mathcal{V} = (\omega, v) \in \mathbb{R}^6$  where  $\omega \in \mathbb{R}^3$  and  $v \in \mathbb{R}^3$  are the angular and linear velocities, respectively.

The *LaserID* only applies for laser scan messages to denote from which sensor the data is read. The data, in this case, consist of 540 range readings from the specified laser.

Similarly, a message of type *setTwist* can be sent, commanding a change in the velocity of the robot.

#### A. Sunrise Application

The *KMRiwaSunriseApplication* is the main application running on the Sunrise Cabinet. The Java classes communicating with ROS2 are initiated from the application as threads and executed in parallel. The Java classes are responsible for sending sensor data and other relevant information, and for carrying out the pose or velocity commands received from the ROS2 nodes. Support classes are implemented for both

the protocol options TCP and UDP for handling the socket objects and transmitting data. Each of the Java communication classes establishes a socket class to transmit the data to the corresponding ROS2 node.

The *kmp\_sensor\_reader* class is handling both the data from the S300 laser sensors and the odometry data. An FDI connection, which is used to transmit odometry and laser data, is established between an instance of the class and the Navigation PC. The FDI connection is based on subscriptions, meaning a data type is only sent if a subscription to the data type is created. This makes it possible to subscribe to only laser or odometry data. A data listener class is implemented that, by subscribing to the data of interest, retrieves odometry data and laser data through the FDI connection. Further, the data is transmitted to the two corresponding ROS2 nodes.

The *kmp\_status\_reader* use functionality from KUKA RoboticsAPI to retrieve information from the robot. Examples of information that can be retrieved are whether an emergency stop has been triggered and if there are any obstacles in the monitored fields.

The *kmp\_commander* receives velocity or pose commands from the corresponding ROS2 node. Pose commands are executed by the motion type *MobilePlatformRelativeMotion* from KUKA RoboticsAPI, while velocity commands are being carried out by jogging the robot. More specific this is done by a *KMPJogger* object, which is implemented to handle the execution of motion and the accumulation of threads.

#### B. Remote PC

On the remote PC, there are four different nodes available for communication with the KMP: *kmp\_odometry*, *kmp\_laser*, *kmp\_command* and *kmp\_status*.

All the nodes have data process methods and ROS publishers and subscribers to correctly handle the data to and from the KMP. All the publishers and subscribers, and the associated ROS topic are listed in Table I and II.

The *kmp\_odometry* and *kmp\_laser* nodes handle the sensor information retrieved from the KMP, while the *kmp\_command* subscribes to multiple ROS topics and forwards the commands from each topic. The currently supported commands are: move with a certain velocity, move to a given pose, and shutdown. The shutdown command is essential to be able to shut down all connections and threads in the program correctly.

The *kmp\_status* retrieves information data from the KMP and saves the information to a ROS message, *KmpStatusdata*. This message type is custom made for the interface and includes information that is useful when operating the robot. The message could be extended with more information if necessary. The *KmpStatusdata* includes the information shown in Table III.

## VI. SYSTEM VERIFICATION USING CARTOGRAPHER AND NAVIGATION2

The ROS packages *Cartographer* and *Navigation2* are used to verify if the communication and control work as expected.

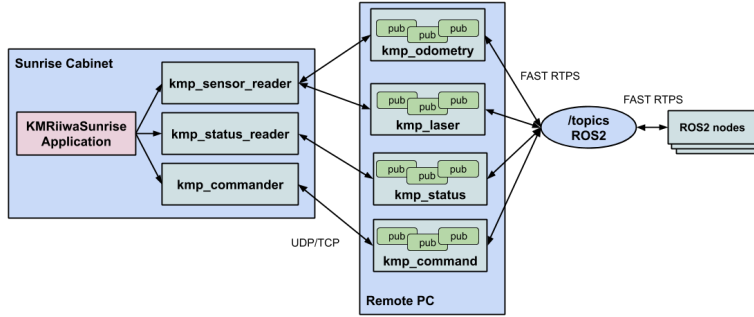


Fig. 3. Architecture of the implemented solution.

TABLE I  
PUBLISHERS FOR PUBLISHING DATA FROM KMP TO ROS

Name	Message type	Topic	Description
pub_odometry	Odometry	/odom	Odometry information.
pub_laserscan1	LaserScan	/scan_1	Data from B1 S300 laser (front).
pub_laserscan2	LaserScan	/scan_2	Data from B4 S300 laser (back).
pub_kmp_statusdata	KmpStatusdata	/kmp_statusdata	Statusdata retrieved in the <i>kmp_status</i> node.

TABLE II  
SUBSCRIBERS FOR SUBSCRIBING TO DATA FROM ROS TO KMP

Name	Message type	Topic	Description
sub_twist	Twist	/cmd_vel	Make KMR move at a certain velocity.
sub_pose	Pose	/pose	Make KMR move to a certain pose.
sub_shutdown	String	/shutdown	Make the application on the Sunrise controller shutdown. Any string sent to this topic do the same purpose.

TABLE III  
FIELDS INCLUDED IN A *KmpStatusdata* MESSAGE

Name	Message type	Description
header	std_msgs/Header	Regular header for all ROS messages.
operation_mode	String	The KMR iiwa has three different operation modes. This field states the current mode.
ready_to_move	Boolean	True if the robot is ready to move, and no safety rules is violated.
warning_field_clear	Boolean	False if either of the warning fields of the S300 sensors are violated.
protection_field_clear	Boolean	False if either of the protection fields of the S300 sensors are violated.
is_kmp_moving	Boolean	True if the KMP is moving.
kmp_safetystop	Boolean	True if the KMP performs a safety stop. This happens if any of the internal safety monitoring functions of Sunrise software are violated.

Both packages include parameter files with multiple parameters that can be tuned to improve the algorithms used. These parameters are minimally tuned for this experiment, and only the parameters necessary for the packages to work with our data have been changed. All available data sources are used,

which includes sensor data from both S300 sensors as well as odometry data.

By driving the robot around in the laboratory and using Cartographer, the map in Figure 4 was created. Tuning the parameters to a more significant extent would likely improve the result, but was not relevant for testing the communication

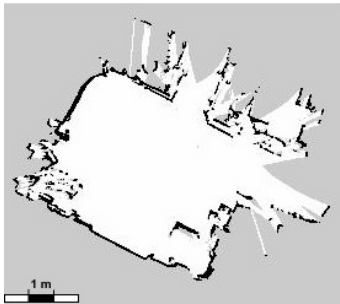


Fig. 4. Map created by Cartographer

interface. As mentioned in section II, the SICK scanners perform a planar scan 150 mm above the ground. This affects the map, as overhanging obstacles are not detected. Hence, additional sensors should be included in the system to be able to perform autonomous navigation safely.

The compatibility with Navigation2 was verified by entering a goal pose in Rviz to which the KMP was to navigate. The goal pose is sent to Navigation2 which returns velocity commands. The use of Navigation2 works as expected for more straightforward scenarios. When commanded to navigate to poses in open areas, the robot moves with holonomic movement and follows the planned path until the requested goal pose. When the goal pose is set too close to obstacles, the navigation is not completed.

The maximum velocity specified in the parameter file of Navigation2 are above 0.13 m/s, which is the velocity where the sensors are activated for T1 mode. If the sensors detect an object inside the protective field when driving at a speed higher than this, the safety restriction of the robot turns in and stops the movement. This causes the navigation to fail, as the vehicle is not following the given commands and not showing enough progress within a time limit. The vehicle is not able to drive out of this area as all the commands from Navigation2 are at a higher speed than allowed by the robot.

When the KUKA Navigation Solution controls the vehicle, the velocity is automatically reduced when objects are inside the protective field. Navigation2 does not implement this behavior, and this causes a conflict between the built-in safety restrictions and the navigation controller. To make Navigation2 work optimally, this must be taken into account. Newly implemented behavior in Navigation2 makes it possible to update the parameters which specify the maximum velocities dynamically.

Our suggested solution to the navigation problem is to monitor the status of the warning fields and protective fields, which both are included in the *KmpStatusdata* message, and use this information to control the velocity. For both operation modes, the robot stops if an obstacle is detected in the

protective field. The velocity should be reduced when an obstacle is detected in the warning field to reduce the size of the protective field. If necessary, a second warning field can be configured by the use of SICK software, to be notified about obstacles earlier. This could solve the problem, but would lead to a less generic system.

## VII. CONCLUSION

This paper describes a control interface for operating the mobile robot KMR iiwa with ROS2. The proposed architecture is verified by a proof-of-concept implementation that enables control of the robot from an external computer. The architecture is based on multiple ROS2 nodes with corresponding Java classes that handle separate tasks. The architecture is created in a scaleable manner, where more nodes can be added for additional functionality.

It was desired to navigate the robot by utilizing the ROS2 packages Cartographer and Navigation2. For this purpose, functionality was implemented to retrieve odometry and laser data from the sensors, and for the ability to control the model by velocity commands. The specified ROS2 packages were used to verify the controllability of the mobile vehicle. The Cartographer package was able to create a fully recognizable map of the environment, and simple navigation in the environment was performed. The verification revealed issues related to navigation closer to obstacles, and a possible solution for this was proposed.

Further work include manipulation of the LBR iiwa, improved navigation, and the addition of camera sensors to handle issues related to 2D laser scans.

The work is open source and available online at [https://github.com/ninamwa/kmriiwa\\_ws](https://github.com/ninamwa/kmriiwa_ws).

## REFERENCES

- [1] KUKA. *KMR iiwa omniMove*, 2019. <https://xpert.kuka.com/app/portal>.
- [2] Direct Industry. *KMR IIWA*. <https://www.directindustry.com/prod/kuka-ag/product-17587-1714901.html>, 2016. The image is used with permission from KUKA Nordic AB. Accessed: 2020-03-13.
- [3] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.
- [4] Salvatore Virga and Marco Eposito. IFL-CAMP/Iiwa Stack. [https://github.com/IFL-CAMP/Iiwa\\_stack](https://github.com/IFL-CAMP/Iiwa_stack), 2019. Accessed: 2019-11-07.
- [5] Saied Mokaram, Jonathan M. Aitken, Uriel Martinez-Hernandez, Iveta Eimontaite, David Cameron, Joe Rolph, Ian Gwilt, Owen McAree, and James Law. A ROS-integrated API for the KUKA LBR iiwa collaborative robot. *IFAC-PapersOnLine*, 50(1):15859 – 15864, 2017. 20th IFAC World Congress.
- [6] Michael Kaßberger Manuel Brucker Tim Bodenmüller Andreas Dömel, Simon Kriegel and Michael Suppa. Toward fully autonomous mobile manipulation for industrial environments. *International Journal of Advanced Heggem Systems*, 2017.
- [7] Charlotte Heggem and Nina Marie Wahl. Project repository: kuka\_ws. [https://github.com/ninamwa/kmriiwa\\_ws](https://github.com/ninamwa/kmriiwa_ws), 2019.
- [8] W. Hess, D. Kohler, H. Rapp, and D. Andor. Real-time loop closure in 2D LIDAR SLAM. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.
- [9] ROS2 Navigation. <https://ros-planning.github.io/navigation2/>. Accessed: 2020-03-27.
- [10] Cartographer. <https://opensource.google/projects/cartographer>. Accessed: 2020-03-27.



# Appendix C.

## Github Repository

The *kmriiwa\_ws* repository, created as a part of the work of this thesis, is available at [https://github.com/ninamwa/kmriiwa\\_ws](https://github.com/ninamwa/kmriiwa_ws).

### C.1. Hierarchy

The software is structured by the following hierarchy:

kmriiwa\_ws

- kmr\_behaviortree
  - behaviortrees
  - include
  - launch
  - param
  - plugins
  - src
- kmr\_bringup
  - launch
  - meshes
  - rviz
  - scripts
  - urdf
- kmr\_communication
  - launch
  - nodes
  - param
  - script
- kmr\_manipulator
  - launch
  - model
  - nodes
  - param
  - script

- kmr\_moveit2
  - config
  - launch
  - rviz
  - src
- kmr\_msgs
  - action
  - msg
- kmr\_navigation2
  - launch
  - map
  - param
  - rviz
  - scripts
- kmr\_simulation
  - launch
  - models
  - worlds
- kmr\_slam
  - config
  - launch
  - rviz
- kmr\_sunrise
  - app
  - comm
  - motion
  - nodes
  - utilities

## C.2. kmriiwa\_ws

Manage topics

305 commits

5 branches

0 packages

0 releases

1 environment

2 contributors

Apache-2.0

Branch: eloquent

New pull request

Create new file

Upload files

Find file

Clone or download

ninamwa Update README.md		Latest commit 5f1e5e9 now
kmr_behaviortree	Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa_ws into...	23 hours ago
kmr_bringup	Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa_ws into...	23 hours ago
kmr_communication	Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa_ws into...	23 hours ago
kmr_manipulator	Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa_ws into...	23 hours ago
kmr_moveit2	Merge remote-tracking branch 'origin/eloquent' into eloquent	3 days ago
kmr_msgs	Update README.md	13 days ago
kmr_navigation2	Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa_ws into...	23 hours ago
kmr_simulation	Merge remote-tracking branch 'origin/eloquent' into eloquent	3 days ago
kmr_slam	Merge remote-tracking branch 'origin/eloquent' into eloquent	3 days ago
kmr_sunrise	cleanup in Java code	4 days ago
.gitignore	nuc files and cartographer tuning	20 days ago
LICENSE	Create LICENSE	4 months ago
README.md	Update README.md	now

README.md

kmriiwa\_ws

Repository for master thesis and specialization project in Robotics & Automation, at NTNU Fall 2019, by Charlotte Heggem and Nina Marie Wahl.

**Intention:** This project aims to create a communication API between a KUKA robot, KMR iiwa, and ROS2. Multiple ROS packages are used for including functionality. Navigation2 is used for navigating the mobile vehicle. Cartographer and RTAB-Map is used for SLAM. MoveIt2 is used for path planning for the manipulator.

Multiple Intel Realsens D435 cmeras are used to provide better moving and detection of objects. A Robotiq gripper is used for picking up objects. The cameras and gripper are launched at a separate onboard computer (Intel NUC).

**System requirements:**

- Ubuntu 18.04.3
- Python 3.6.9
- ROS Eloquent

**Required ROS Packages:**

- Gazebo packages
- Navigation2
- MoveIt2
- Cartographer
- RTAB-Map ROS wrapper (dependent on RTAB-Map)
- Ros2 Intel Realsense (ROS2 Wrapper for Intel® RealSense™ Devices)
- ROS2 Openvino Toolkit (dependent on OpenVino Toolkit)
- ROS2 Object Analytics

### C.3. kmr\_behaviortree

Branch: eloquent kmriiwa\_ws / kmr\_behaviortree /

Create new fileUpload filesFind fileHistory

Merge branch 'eloquent' of https://github.com/ninemwa/kmriiwa\_ws into... Latest commit 25d8549 20 hours ago

..

behavior\_trees

Merge branch 'eloquent' of https://github.com/ninemwa/kmriiwa\_ws into...20 hours ago

include/kmr\_behaviortree

working mani moving to objectlast month

launch

cleanup NUC20 hours ago

param

added new workstations poses and small cleanup4 days ago

plugins

added new workstations poses and small cleanup4 days ago

src

added new workstations poses and small cleanup4 days ago

CMakeLists.txt

renaming of nodes + making kmr drive back home when finishedlast month

README.md

Create README.md12 days ago

package.xml

merge branch behaviortree into branch eloquent2 months ago

README.md

### 1. Description

This package includes functionality for using a behavior tree to create more complex functionality for the robot. The plugin folder consists of custom made condition and action nodes. These nodes can be put together in multiple ways to create a full tree. Different behavior trees are found in the behavior\_trees folder, and which tree to use is specified in the launch file. The param file can be used to specify different information, like positions of workstations and a list of goals. The behaviortree will start over for every goal pose in the list. When the list is empty, the robot will navigate back to the home position/docking station.

All other nodes which the behavior tree nodes depends upon must be running for the behavior tree to be initialized.

### 2. Requirements

The following packages needs to be installed:

- behaviortree\_cpp\_v3

### 3. Run

To launch the behaviortree, run:

```
$ ros2 launch kmr_behaviortree bt.launch.py
```

When the behavior tree is initialized, a message to the /start\_topic must be sent for the execution of the tree to start. This is done to make sure everything is correctly set up before starting, and should be removed at a later point.

```
$ ros2 topic pub /start_topic std_msgs/msg/String {'data: OK'} -1
```



## C.4. kmr\_\_bringup

Branch: eloquent


kmriiwa\_ws / kmr\_bringup /

Create new file

Upload files









Find file


History

 Merge branch 'eloquent' of [https://github.com/ninamwa/kmriiwa\\_ws](https://github.com/ninamwa/kmriiwa_ws) into...

...

Latest commit 25d8549 20 hours ago

..		
 launch	cleanup NUC	20 hours ago
 meshes	full renaming from kuka to kmr	2 months ago
 rviz	nuc files and cartographer tuning	20 days ago
 scripts	working mani moving to object	last month
 urdf	added gripper adapter link	4 days ago
 CMakeLists.txt	full renaming from kuka to kmr	2 months ago
 README.md	Update README.md	8 days ago
 package.xml	full renaming from kuka to kmr	2 months ago

 README.md

### 1. Description

This package handles the different files for bringing up the robot and showing it in Rviz.

### 2. Requirements

The following packages needs to be installed:

- joint\_state\_publisher
- robot\_state\_publisher

### 3. Run

To visualize the URDF model of the robot in Rviz, you need two terminals and run the following commands:

```
$ ros2 launch kmr_bringup rviz.launch.py
```

```
$ ros2 launch kmr_bringup state_publisher.launch.py
```


```
$ ros2 run kmr_bringup dummy_joint_states
```

The latter will run a dummy joint state publisher which publishes fake data for the joints which are not fixed. This is necessary to properly visualize the manipulator.

## C.5. kmr\_communication

Branch: eloquent kmriiwa\_ws / kmr\_communication /

Create new fileUpload filesFind fileHistory

 Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa\_ws into... Latest commit 25d8549 20 hours ago

..

launch

Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa\_ws into...

20 hours ago

nodes

Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa\_ws into...

20 hours ago

param

code cleanup, small fixes

4 days ago

script

code cleanup, small fixes

4 days ago

CMakeLists.txt

merge branch behaviortree into branch eloquent

2 months ago

README.md

Update README.md

13 days ago

package.xml

merge branch behaviortree into branch eloquent

2 months ago

README.md

### 1. Description

This package handles the communication with the KMR iiwa. There is one launch file which will launch all of the seven communication nodes:

- kmp\_commands\_node
- kmp\_odometry\_node
- kmp\_laserscan\_node
- kmp\_statusdata\_node
- lbr\_statusdata\_node
- lbr\_commands\_node
- lbr\_sensordata\_node

The connection type (UDP/TCP) can be set in the launch parameters. The IP address to your computer should be set in the parameter file called bringup.yaml. In this file, you can also change the port number for each of the nodes.

### 2. Run

To launch all of the communication nodes, run:

```
$ ros2 launch kmr_communication sunrise_communication.py
```

# C.6. kmr\_manipulator

Branch: eloquent kmriiwa\_ws / kmr\_manipulator /

Create new fileUpload filesFind fileHistory

Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa\_ws into... Latest commit 25d8549 20 hours ago

..

launch

Merge branch 'eloquent' of https://github.com/ninamwa/kmriiwa\_ws into...20 hours ago

model

objectdetection node updateslast month

nodes

cleanup NUC20 hours ago

param

code cleanup, small fixes4 days ago

script

fix activation status and compability between gripper and NUC. Fix la...last month

CMakeLists.txt

renaming of nodes + making kmr drive back home when finishedlast month

README.md

Update README.md7 days ago

package.xml

full renaming from kuka to kmr2 months ago

README.md

1. Description

This package handles tasks associated with manipulation of the LBR iiwa.

- Vision using a Intel® RealSense™ D435 camera
- Object detection and localization
- Grasping using a Robotiq 2F-85 gripping

2. Requirements

The following packages needs to be installed:

- ROS2 Intel Realsense
- ROS2 Openvino Toolkit
- ROS2 Object Analytics

3. Run

The camera and gripper must be connected to a computer by USB. An onboard computer with ROS2 installed is useful for this puprose. The nodes are launched by running the command:

```
$ ros2 launch kmr_manipuulator nuc.launch.py
```

# C.7. kmr\_\_moveit2

Branch: eloquent kmriiwa\_ws / kmr\_moveit2 /

Create new fileUpload filesFind fileHistory

ninamwa Merge remote-tracking branch 'origin/eloquent' into eloquentLatest commit 88a5da5 3 days ago

..

config	change order of search 2 and 3	3 days ago
launch	code cleanup, small fixes	4 days ago
rviz	full renaming from kuka to kmr	2 months ago
src	code cleanup, small fixes	4 days ago
.project	full renaming from kuka to kmr	2 months ago
CMakeLists.txt	renaming of nodes + making kmr drive back home when finished	last month
README.md	Create README.md	5 days ago
package.xml	merge branch behaviortree into branch eloquent	2 months ago

README.md

### 1. Description

This package handles the use of the MoveIt2 package for path planning for the LBR iiwa manipulator.

### 2. Requirements

The following packages needs to be installed:

- MoveIt2

### 3. Run

Run the following command to launch MoveIt:

```
$ ros2 launch kmr_moveit2 moveit.launch.py
```

MoveIt can be used in three different ways:

- Through the action PlanToFrame.
- By publishing a ROS PoseStamped to the /moveit/goalpose topic
- By publishing a ROS String describing a configured frame to the /moveit/frame2 topic. The possible frames are described in the SRDF file.

C.8. kmr\_msgs

Branch: eloquent


kmriiwa\_ws / kmr\_msgs /

Create new file

Upload files


Find file

History

 Update README.md


Latest commit 0f6e545 13 days ago

..

 action


[renaming of nodes + making kmr drive back home when finished](#)

last month

 msg


working mani moving to object

last month

 CMakeLists.txt


renaming of nodes + making kmr drive back home when finished

last month

 README.md


Update README.md

13 days ago

 package.xml

rebuilt to own kmr\_msgs folder

2 months ago

 README.md


1. Description

This package includes the different set of messages, services and actions which are customized for the kmriiwa workspace.

# C.9. kmr\_navigation2

Branch: eloquent kmrriwa\_ws / kmr\_navigation2 /









Create new fileUpload filesFind fileHistory


ninamwa

Merge branch 'eloquent' of [https://github.com/ninamwa/kmrriwa\\_ws](https://github.com/ninamwa/kmrriwa_ws) into...

...

Latest commit 25d8549 20 hours ago

..		
 launch	code cleanup, small fixes	4 days ago
 map	small naming fixes	12 days ago
 param	Merge branch 'eloquent' of <a href="https://github.com/ninamwa/kmrriwa_ws">https://github.com/ninamwa/kmrriwa_ws</a> into...	20 hours ago
 rviz	cleanup NUC	20 hours ago
 scripts	Merge branch 'eloquent' of <a href="https://github.com/ninamwa/kmrriwa_ws">https://github.com/ninamwa/kmrriwa_ws</a> into...	20 hours ago
 CMakeLists.txt	keyboard which changes the speed based on warning field	last month
 README.md	Update README.md	13 days ago
 package.xml	full renaming from kuka to kmr	2 months ago

 README.md

### 1. Description

This package handles the use of the Navigation2 package for autonomous control of the KMR iwa robot.

### 2. Requirements

The following packages needs to be installed:

- Navigation2

### 3. Run

You need two terminals where you are running the commands:

```
$ ros2 launch kmr_communication sunrise_communication.launch.py
```

```
$ ros2 launch kmr_navigation2 navigation2.launch.py
```

This will launch the communication nodes for communicating with the KMR iwa, and the navigation stack. In Rviz, you need to set the initial pose of the robot by pressing the "2D Pose Estimate" button. The robot should get some time to localize itself. Drive it a around by using a keyboard. This can be launched by running:

```
$ ros2 run kmr_navigation2 twist_keyboard.py
```

This keyboard send velocity commands to the robot. The robot can also be navigating by using a pose keyboard, where you are giving a desired pose of the robot. This can be launched by running:

```
$ ros2 run kmr_navigation2 pose_keyboard.py
```

When you are ready to start the navigation, press the "Navigation2 Goal" button, and set the goal target in the map.

The robot can also be navigating by using a pose keyboard, where you are giving a desired pose of the robot. This can be launched by running:

A keyboard for manually controlling both the mobile vehicle and manipulator at the same time can be launched by running:

```
$ ros2 run kmr_navigation2 keyboard.py
```

# C.10. kmr\_simulation

Branch: eloquent


kmriiwa\_ws / kmr\_simulation /

Create new file







Upload files


Find file


History

 **ninamwa** Merge remote-tracking branch 'origin/eloquent' into eloquent

Latest commit 88a5da5 3 days ago

..		
 <a href="#">launch</a>	full renaming from kuka to kmr	2 months ago
 <a href="#">models/kmr</a>	small naming fixes	12 days ago
 <a href="#">worlds</a>	full renaming from kuka to kmr	2 months ago
 <a href="#">CMakeLists.txt</a>	full renaming from kuka to kmr	2 months ago
 <a href="#">README.md</a>	Update README.md	5 days ago
 <a href="#">package.xml</a>	full renaming from kuka to kmr	2 months ago

 [README.md](#)



### 1. Description

This package is used for simulating the robot in Gazebo.

### 2. Requirements

The Gazebo software and the ROS packages for interfacing with Gazebo, called gazebo\_ros\_pkgs, must be installed.

### 3. Run

To start up Gazebo, run:

```
$ ros2 launch kmr_simulation gazebo.launch.py
```

In addition, you need to launch the robot\_state\_publisher:

```
$ ros2 launch kmr_bringup state_publisher.launch.py
```

This will launch a model of the robot in Gazebo, and it is possible to control it, by using the twist keyboard:

```
$ ros2 run kmr_navigation2 twist_keyboard.py
```

The keyboard will make the robot move around in the simulated environment. Other packages like SLAM and Navigation may also be used together with Gazebo!

# C.11. kmr\_slam


Branch: eloquent | kmriiwa\_ws / kmr\_slam /

Create new file

Upload files








Find file

History

ninamwa

Merge remote-tracking branch 'origin/eloquent' into eloquent

Latest commit 88a5da5 3 days ago

..		
	config	Final maps and SLAM configurations. Increase range of laserscan range... 15 days ago
	created_maps	code cleanup, small fixes 4 days ago
	launch	Merge branch 'eloquent' of https://github.com/ninamwa/kuka_ws into el... 15 days ago
	rviz	nuc files and cartographer tuning 20 days ago
	CMakeLists.txt	renaming from kmr_cartographer to kmr_slam last month
	README.md	Update README.md 5 days ago
	package.xml	renaming from kmr_cartographer to kmr_slam last month

README.md

### 1. Description

This package is used for performing realtime SLAM and creating a map together with the KMR iiwa. Both the packages Cartographer and RTAB-Map can be used.

### 2. Requirements

The following packages needs to be installed:

- Cartographer
- Cartographer\_ros
- RTAB-Map
- rtabmap\_ros (ROS wrapper for RTAB-Map)
- Nav2\_map\_server (for saving the maps - it is a part of the Navigation2 package)

### 3. Run

To launch Cartographer, run:

```
$ ros2 launch kmr_slam cartographer.launch.py
```

To launch RTAB-Map, run:

```
$ ros2 launch kmr_slam rtabmap.launch.py
```

The communication with the KMR must be started, and the robot can be driven around manually by using the implemented keyboard:

```
$ ros2 run kmr_navigation2 twist_keyboard.py
```

If you want to save the map which are created, this can be done by running the following command in a separate terminal:

```
$ ros2 run nav2_map_server map_saver
```



## C.12. kmr\_sunrise

Branch: eloquent kmriiwa\_ws / kmr\_sunrise /

Create new fileUpload filesFind fileHistory

charlotteheggem

cleanup in Java code

Latest commit 0363d0f 4 days ago

..		
app	cleanup in Java code	4 days ago
comm	cleanup in Java code	4 days ago
motion	cleanup in Java code	4 days ago
nodes	cleanup in Java code	4 days ago
utilities	cleanup in Java code	4 days ago
README.md	Update readme with dependency	4 days ago

README.md

### 1. Description

This package contains the Java program that is to be installed and launched on the Sunrise Cabinet.

In addition to the main application, KMRIiwaSunriseApplication, the package include the following communication nodes:

- kmp\_commander
- kmp\_sensor\_data
- kmp\_status\_data
- lbr\_commander
- lbr\_sensor\_data
- lbr\_status\_data
- lbr\_commander

The Javadocs of the package can be found at [https://ninamwa.github.io/kmriiwa\\_ws/](https://ninamwa.github.io/kmriiwa_ws/)

The connection type (UDP/TCP) and port can be set for each node in the KMRIiwaSunriseApplication. The IP address to the remote computer must be defined in either of the TCPsocket or UDPsocket classes, depending on the choice of protocol.

The files must be downloaded to a Sunrise project and synchronized to the controller from Sunrise Workbench.

### 2. Requirements

In addition to the default KUKAJavaLib, the following .jar packages must be added to the library of the project:

- com.kuka.common
- com.kuka.nav.comm.api
- com.kuka.nav.encryption.provider.api
- com.kuka.nav.provider
- com.kuka.nav.provider
- com.kuka.robot.fdi.api
- bjprov-djk15on-154
- log4j
- slf4j.api
- slf4j.log4j12

### 2. Run

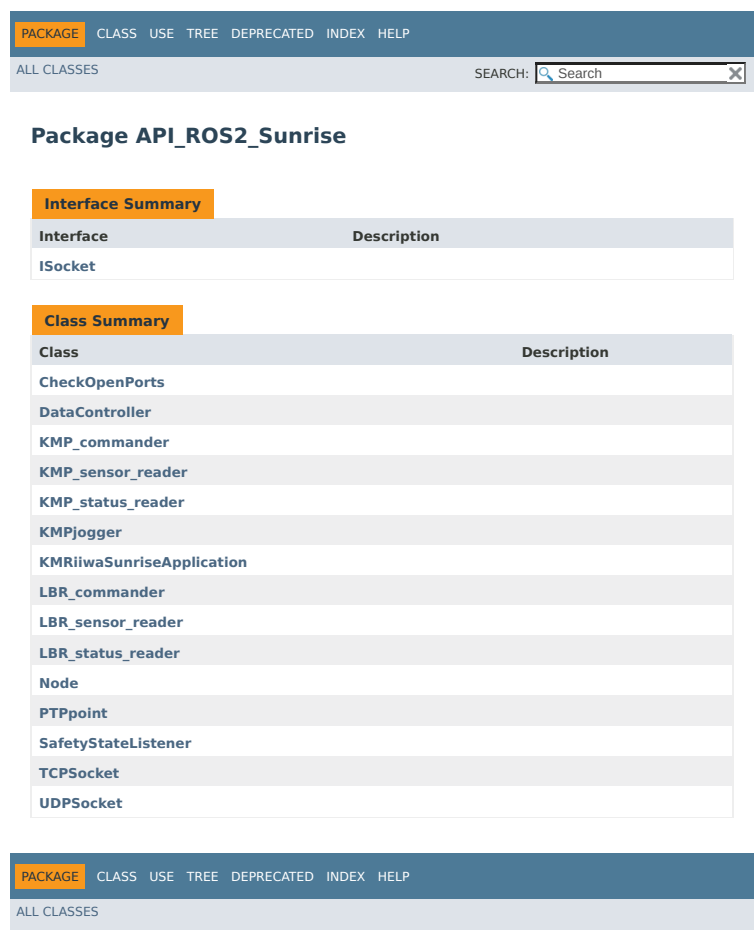
The KMRIiwaSunriseApplication can be launched from the smartPAD when the project is installed on the Sunrise Cabinet.



# Appendix D.

## Javadoc

A javadoc is generated for the Java implementation of the system, which can be found at [https://ninamwa.github.io/kmriiwa\\_ws/](https://ninamwa.github.io/kmriiwa_ws/). The documentation contains an overview of the implemented classes and how they are related. Figure D.1 shows the overview page of the package, containing the implemented Java classes and interfaces. Further, each class contains nested classes, constructors, fields and methods that can be inspected by navigating the javadocs. An example of a class page is shown in Figure D.2 for the abstract class Node.



**Figure D.1.:** Javadoc overview of the package `API_ROS2_Sunrise` with the implemented Java classes.

PACKAGECLASSUSE TREEDEPRECATEDINDEXHELP

ALL CLASSES

SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHODDETAIL: FIELD | CONSTR | METHOD

Package API\_ROS2\_Sunrise

Class Node

java.lang.Object  
  java.lang.Thread  
    API\_ROS2\_Sunrise.Node

All Implemented Interfaces:  
java.lang.Runnable

Direct Known Subclasses:  
KMP\_commander, KMP\_sensor\_reader, KMP\_status\_reader, LBR\_commander, LBR\_sensor\_reader, LBR\_status\_reader

public abstract class Node  
extends java.lang.Thread

Nested Class Summary

Nested classes/interfaces inherited from class java.lang.Thread

java.lang.Thread.State, java.lang.Thread.UncaughtExceptionHandler

Field Summary

Fields

Modifier and Type	Field	Description
boolean	closed	
static int	connection_timeout	

Fields inherited from class java.lang.Thread

MAX\_PRIORITY, MIN\_PRIORITY, NORM\_PRIORITY

Constructor Summary

Constructors

Constructor	Description
Node(int port1, java.lang.String Conn1, int port2, java.lang.String Conn2, java.lang.String nodeName)	
Node(int port, java.lang.String Conn, java.lang.String nodeName)	

Method Summary

All MethodsStatic MethodsInstance MethodsAbstract MethodsConcrete Methods

Modifier and Type	Method	Description
abstract void	close()	
void	createSocket()	
void	createSocket(java.lang.String Type)	
boolean	getEmergencyStop()	
boolean	getisKMPConnected()	

Figure D.2.: Example of a javadoc page for the abstract class Node.

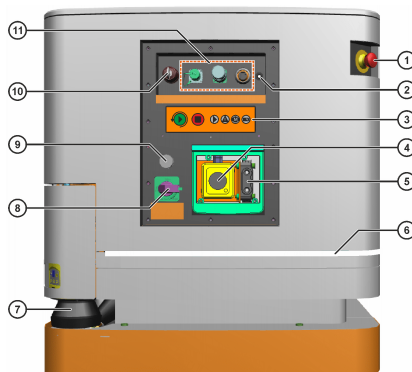


## Appendix E.

# Operating the KMR

This appendix includes instructions for operating the KMR. The content is reproduced based on the specialization project, and is included for users who intend to use the developed system to control the KMR. First, different components, indicators and switches on the KMR is presented. Next, a short description of the smartPAD and how to operate it is given. Next follows a section on how to launch an application on the Sunrise Cabinet.

The rear side of the KMP contains a panel with components for operating the vehicle.



**Figure E.1.:** The rear of a KMP 200 omniMove [23, p. 18].

The elements denoted by circled numbers in figure E.1 will be explained. The main switch (8) is used to turn on the KMR. The key switch (10) has three positions. The first is used to operate the KMR with the radio control unit. To operate the KMR with an application, the key switch must be set to the second position. The third position is used to release the brakes for emergency operation of the vehicle. The device for releasing the brakes can be connected at (11). To use the



**Figure E.2.:** The KUKA smartHMI user interface on the smartPAD [23, p. 141].

release device, the robot must be switched off, and the robot can be pushed when holding the button at the releasing device.

The emergency stop button ① can be used to stop the KMR. There are two such buttons at the KMP, one at the front and one at the rear. The panel ③ contains display elements that indicate the states of the robot during operations. The platform has a LED strip denoted ⑥ that signals defined states in different colors. The state indicators can be used for troubleshooting. When stationary, the KMR can be charged when by inserting the charger to the socket ⑤ from the external battery charger. The smartPAD can be connected at ⑪.

## E.1. SmartPAD

Figure E.2 shows the smartHMI on the smartPAD. An application can be selected by choosing the level for Applications in the navigation bar ①. The synchronized Sunrise project with the available applications will appear in a list. Sunrise Projects can be synchronized to the Sunrise Cabinet via Sunrise Workbench. In Figure E.2, the tab is grey, indicating that no application is chosen. When an application is chosen, the display area ⑤ changes to a different view.



When the motion is enabled, the application is launched by the green button on the rear of the smartPAD. The yellow status indicator for the Applications level indicates that an application is paused. Logged information and eventual errors from launching an application are shown in the display area ⑤.

The Robot level in ① displays the selected platform, and the display area ⑤ provides information about the chosen robot. The chosen robot can be moved, or jogged, manually by pressing the jog keys on the left side of the smartPAD. Operation mode must be set to T1, and the motion must be enabled.

The operating mode of the system can be changed by turning the key switch on the smartPAD and choosing the desired mode. The smartPAD has three enabling switches located on the back of the device. All the switches are white and have three positions: not pressed, fully pressed, and center position. The motion is enabled if one of the buttons is held at the center position, and the robot can be moved manually. An emergency stop button is located on the front side of the smartPAD.

## E.2. Signal Units

The KMP has signal units to indicate the states of the robot, the operator panel, the LED strip and the display on the SICK scanners, which are useful for troubleshooting. Common faults are listed in chapter 12 in the documentation of the KMR [23, p. 222]. The display elements on the operator panel provide information about the state of the vehicle [23, p. 117]. The LED strip around the KMP indicates the operating state of the vehicle. The different types of signals are described in the operating manual of the KMR [23, p. 31]. Information about the SICK laser scanners can be found in the manual [23, p. 224], and different signals for the display of the panel can be found in the operating instructions for the SICK S300 [28, p. 117].

## E.3. Launching an Application

This description assumes the complete project is installed, with the associated parts on the work computer and the remote computer. The remote computer must be connected to the ASUS 5G network, which is the network used by the Sunrise Cabinet. The IP address and ports defined for TCP or UDP connections must be set in the corresponding Java socket class and Sunrise application, respectively. The same information needs to be specified in the bringup configuration file in `kmr_communication`.

The KMR is turned on by turning the main switch on the rear of the KMP from 0 to 1, denoted ⑧ in figure E.1. After a few seconds, the smartPAD is enlightened, and the startup of the OS is initialized. When the smartHMI shows the station display as in figure E.2, the configuration is complete. A yellow status indicator is shown next to the station, indicating a safety warning that concerns the KMP. This can be ignored.

If the platform is to be operated in manual mode, it is necessary to be two persons. One person needs to handle the Sunrise smartPAD to activate motion, while the other person controls the remote computer. It is advised that the system is started with no obstacles close to the KMR. The person holding the smartPAD should keep a distance from the vehicle when launching the application. The application is selected by choosing the Application tab in the navigation menu. Further, the white enable button must be pressed, followed by the green start button. Both buttons are located on the back of the smartPAD. The buttons must be held through the entire execution to enable the motion of the vehicle.

If the platform is to be operated in autonomous mode, the Java program *PositionAndGMSReferencing* needs to be launched. The application can be selected from the application drop-down menu on the smartPAD. This requires that the robot is in the operation mode T1. When the program has terminated, the key on the smartPAD can be switched to select AUT mode. Further, the application can be selected as described for manual modes. The green start button on the smartPAD is used to execute the application.

