Electrical &
Electronic
Engineering.

Human-Robot Collaboration Furniture Assembly with Integrated Safety Protocols

Quang Hung Tran

September, 2025

Supervisor: Dr Jonathan Aitken

A dissertation submitted in partial fulfillment of the requirements for the degree of MSc in Robotics.

ABSTRACT

Assembling internal and external furniture by collaborative robots (cobots), robots that work alongside humans, is increasingly employed in many countries and in everyday life, aiming to reduce labor costs, avoid injury, and increase productivity. However, the safety of collaboration between humans and robots is vulnerable and fragile, posing a threat to both humans and robots, such as serious injuries or occupational accidents. To overcome this challenge, a new cobot system is introduced in this project to enhance safety, work speed, flexibility, and productivity, with a primary focus on improving the safety level while maintaining the precision and accuracy of the assembly tasks. This system will help the robot understand its surroundings, plan actions, and interact smoothly with humans. ISO 10218-1/2 and ISO/TS 15066 established the groundwork for the study. Building upon this, the robot operating system (ROS) and Gazebo were utilized to simulate collaborative activities, such as placing pegs into holes or assembling an IKEA chair. Meanwhile, different levels of safety are used when humans enter and leave the workplace. Throughout this study, a high level of safety of collaborative systems is discussed and highlighted, emphasizing the advantages and disadvantages of the safety system. An effective collaborative robot system can bring significant cost savings, reduce labor costs, enhance safety and system reliability, and offer substantial benefits to the assembling industries, especially furniture.

Keywords: *Collaborative robots, furniture assembly, human–robot interaction, safety standards, ISO/TS 15066, ROS, Gazebo simulation, ergonomic safety, motion planning, point cloud processing*

INDIVIDUAL CONTRIBUTION

Working with the Robot Operating System (ROS-Noetic), Gazebo 11, and MoveIt was a central and rewarding part of this project. Blender was also used to support the tasks, such as the IKEA chair [1]. As my first time independently deploying a robotic arm in simulation, it challenged my research skills, self-learning, and adaptability. The system was built around the UR5 manipulator and a Robotiq 2F gripper, using publicly available ROS packages and URDF models [2][3][4][5].

Although my work was limited to simulation, the unpredictability of the virtual environment underscored the importance of flexibility in project management and the need to allocate sufficient time for iterative development. It revealed a fundamental truth about system design: real-world behavior, whether physical or simulated, rarely aligns perfectly with initial expectations. This insight emphasized the necessity of continuous testing and refinement, shaping how I approached the technical aspects of the project and reinforcing the value of adaptability throughout the development process.

The project's path, from planning and working with my supervisor to making final presentations, was critical in helping me improve both my technical and soft skills. It was essential to communicate complex ideas clearly and successfully, particularly when discussing safety rules and the reasoning behind collaborative robots to diverse groups of people.

Some parts of the project were harder than others. Object detection and pick-and-place execution needed more help, calibration, and debugging over time. On the other hand, task design and point cloud processing were done more independently. These segments were carefully thought out, thoroughly tested, and refined with minimal assistance, which ensured my ability to solve problems and integrate systems.

Managing time became a big problem. Some parts of the project are not proceeding as planned, and as a result, more time is required than initially expected. Additionally, there were numerous potential applications of collaborative robots for safe furniture assembly; however, due to time constraints, not all of them could be fully explored, including collaborative strategies. Looking back, a more organized pre-planning phase might have made better use of resources and cut down on delays.

Despite these challenges, the skills and knowledge gained from this project have proven quite useful. They are a great base for future work in safety-critical system design, simulation-driven validation, and collaborative robotics.

ACKNOWLEDGEMENTS

My sincere appreciation goes out to everyone who supported me throughout the course of this dissertation.

Above all, I would like to express my deepest gratitude to my supervisor, Dr. Jonathan Aitken, for his invaluable guidance, technical expertise, and unwavering encouragement. His proficiency in collaborative robotics and safety was instrumental in shaping the direction and scope of this research.

I am also grateful to Nathan Plumb, a PhD student, whose assistance in identifying suitable libraries and troubleshooting code was immensely helpful during the development phase.

I would like to extend my thanks to my peers and the academic staff at the University of Sheffield for fostering a supportive and intellectually stimulating environment. Special appreciation goes to my classmates, whose constructive feedback during presentation rehearsals helped refine and strengthen my work.

I would also like to acknowledge the open-source communities and technical support teams behind tools such as ROS, Gazebo, C++, and Python. Their contributions enabled effective experimentation and model development and played a vital role in the success of this project.

On a personal note, I am deeply grateful to my family and friends for their patience, encouragement, and unwavering support throughout the long hours of writing and research. Their support kept me grounded and motivated.

Ultimately, I dedicate this work to the broader community of engineers and researchers who strive to advance collaborative robots toward a future that is safer, smarter, and more sustainable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter begins by outlining the broader context of industrial challenges and explains the rationale for focusing specifically on the furniture assembly sector and safety, as presented in Section 1.1. Section 1.2 then delves deeper into the underlying issues, offering a detailed background that frames the technical and ergonomic concerns addressed in this study. The subsequent motivation section, 1.3, further articulates the need for this research, highlighting the practical and societal drivers behind the development of a safe and efficient collaborative robotic system. Section 1.4 gives the aims and objectives, while Section 1.5 explains project management and the structure of the dissertation.

1.1 Overview

Manufacturing and assembly industries face increasing pressure to reduce cycle time, improve productivity, and stay competitive while ensuring worker well-being. In furniture assembly, workers are often exposed to repetitive lifting, pushing, and awkward postures, leading to Work-related Musculoskeletal Disorders (WRMSDs), lost productivity, and significant economic costs [6]. Even leading companies like IKEA continue to face ergonomics challenges despite implementing safety initiatives [7].

To prevent this issue, many furniture assembly industries are adopting collaborative robots as a potential solution. By taking over repetitive, hazardous, and precision-demanding tasks. This enables robots to perform continuously at high speed, reduces cycle time, boosts productivity, and minimizes physical strain on workers [9]. Meanwhile, human workers contribute strengths that robots cannot easily replicate, such as adaptability, problem-solving skills, and

decision-making abilities. For this purpose, in recent years, Human-Robot Interaction (HRI) has been widely researched and deployed with a high level of safety [8]. This creates opportunities for effective human-robot collaboration.

However, integrating cobots into shared workspaces raises important safety challenges. Many existing systems rely on basic safeguards, such as collision avoidance and emergency stop, which are often insufficient. When safety is treated as an add-on rather than an integral part of system design, the result is poor coordination and reduced trust between humans and robots. Therefore, humans should be centered in the system design, ensuring that collaborative systems are designed from the outset to prioritize safety alongside productivity [9].

This thesis explores such an approach through the development of a collaborative robotic system for furniture assembly. The research focused on integrating safety protocols within the collaboration framework to demonstrate how humans and robots can work together effectively in complex, real-world assembly tasks.

1.2 Background

Traditionally, humans and robots work separately with a physical barrier to ensure safety and prevent collisions. Collaborative robots, or cobots, enable seamless human-robot interaction in shared workspaces by leveraging advanced sensors and control methodologies [9]. Cobots are increasingly deployed in manufacturing environments to handle repetitive, hazardous, or physically demanding tasks, allowing human workers to focus on adaptive, decision-making, and problem-solving activities [10].

The term “collaborative robot” refers to robots that physically interact with humans or operate in a shared workspace to support human tasks. Industrial cobots are designed to help workers in tasks such as lifting, moving, and assembling materials, aiming to reduce mental and physical workload while improving efficiency and safety. In assembly operations, particularly in the furniture industry, workers are often exposed to Work-related Musculoskeletal Disorders (WRMSDs) and reduced productivity [10].

Ensuring safety in human-robot collaboration is a critical challenge. Recent applications of safety for traditional robots utilize physical or sensor-based barriers to ensure the safety of operators. However, such barriers are eliminated as collaborative robots have their safety systems to keep human workers safe [20]. Although many industrial cobots include basic safety

features such as collision avoidance, emergency stops, and speed limitations, these measures alone may not fully protect workers in dynamic assembly environments. Standards such as ISO 10218-1 and 10218-2, along with ISO/TS 15066, provide guidelines for robot design, installation, and safe human behavior. ISO 10218-1 specifies requirements and designs for industrial robot systems and components, while ISO 10218-2 focuses on robot integration and application into workspaces, including collaborative applications. Meanwhile, ISO/TS 15066 supplements ISO 10218-1/2 for collaborative robots with modes of operation [16][17][18]. However, practical deployment must also account for sensor limitations, blind spots, and real-time decision-making [11].

In furniture assembly, integrating cobots with comprehensive safety protocols is particularly important. Cobots can assist with heavy lifting, precision assembly, and repetitive tasks, while humans contribute flexible, problem-solving based on different situations. Combining these strengths requires systems that actively monitor human presence and access risk and adapt robot behavior accordingly, such as using LiDAR or vision-based monitoring for real-time workspace awareness [10].

1.3 Motivation

Despite advances in industrial automation, furniture assembly remains a physically demanding and ergonomically risky occupation. Workers frequently perform repetitive tasks, such as lifting, pushing, and handling heavy components, which can lead to Work-Related Musculoskeletal Disorders (WRMSDs) and reduced productivity [10]. While collaborating robots (cobots) have been introduced to mitigate these risks, many current implementations rely on basic safety features such as emergency stops and collision avoidance, which are often insufficient in dynamic, real-world assembly environments.

Standards such as ISO 10218-1/2 and ISO/TS 15066 provide guidelines for safe human-robot interaction, yet practical deployments face challenges including sensor limitations, blind spots, and the need for real-time adaptation to human actions. Additionally, most systems are designed with a robot-centric approach, neglecting the holistic integration of human capabilities, task requirements, and safety considerations [9].

This research is motivated by the need to create a human-centered, task-based collaborative system that not only reduces physical strain and improves productivity but also ensures a high

level of safety during furniture assembly. By integrating adaptive sensing technologies, adaptive control strategies, and risk assessment protocols, this study aims to bridge the gap between existing standards and practical industrial implementation. The resulting system will demonstrate how humans and robots can work safely and efficiently together, providing a scalable model for future collaborative manufacturing applications.

1.4 Aims and Objectives

The primary aim of this study is to develop an intelligent collaborative robotic system for automating IKEA furniture assembly. The goal is to optimize efficiency, safety, and adaptability in human-robot collaboration.

The objectives outlined below ensure a reachable and structured pathway, beginning with literature analysis and culminating in the development and simulation of a sophisticated collaborative robotic system tailored for furniture assembly tasks.

1.4.1 Basic Objectives

- Review literature on collaborative robots, cobot safety protocols, and the furniture assembly industry.
- Gain a comprehensive understanding of collaborative robot safety systems, particularly in shared workspaces.
- Design a modular robotic system incorporating vision, motion planning, object handling, and verification processes.
- Simulate a collaborative robotic system to perform pick-and-place operations across two distinct tasks.
  - **Task 1:** Recognize and capture a peg using a 3D camera with point cloud processing, then apply a path planning algorithm to insert the peg into a designated hole.
- Develop a safety system for collaborative robots to ensure safe and effective human-robot interaction during assembly tasks.

1.4.2 Advanced Objectives

- Extend the simulation to include more complex furniture assembly tasks.
  - **Task 2:** Assemble an IKEA chair with randomly positioned components on the ground, sequentially picking and assembling them into a final structure.
- Integrate collaborative assembly operations with real-time safety monitoring and control protocols.
- Evaluate system performance by measuring task duration, assembly accuracy, safety, and the quality of human-robot collaboration.

1.5 Project Management and Dissertation Structure

A structured and manageable approach was essential for managing the time to complete the technical process and dissertation.

At first, the project's aims and objectives were defined differently, and the Gantt Chart was developed accordingly, as illustrated in Figures 1.1–1.5. However, due to time constraints and workload limitations, the scope and objectives were subsequently revised to better align with the available resources and project timeline.

The project was then newly divided into different phases, which contain small objectives and milestones. To track progress, Gantt charts were used, as shown in Figure 1.6. The figures show the planned and actual timeline.

Initially, the strategy was to work through the objectives in chronological order. As research advanced and coding became complex, it became clear that safety and robot-human collaboration tasks needed to be addressed concurrently to address emerging insights and issues. As research progressed, tasks were added to the Gantt Chart, with "Planned" bars, and "Actual" is the real progress of the project.

Completing certain tasks took longer than expected, leading to valuable discoveries. To address this, dissertation chapters were produced simultaneously and started earlier rather than waiting for all duties to be finished. This allowed for the timely completion of the dissertation while exploring new areas.

Effective communication and collaboration were vital. Regular meetings were held once a week to discuss progress, address challenges, and plan next steps. These meetings followed a

systematic approach. All discussions from meetings are meticulously noted in a personal note.

To address the research questions, the ROS-Noetic, Gazebo 11, and MoveIt frameworks were used to simulate and analyze. In addition, Blender was employed to design the objects for pick-and-place tasks. LaTeX made document generation more efficient and consistent, allowing for seamless integration of findings while working on other tasks.

The project was completed successfully thanks to a methodical strategy, efficient communication, and proficient instruments.

The second chapter of this dissertation contains literature surveys. Chapter 3 provides detailed methodologies for the simulation experiments. Chapter 4 is results and discussion, and Chapter 5 provides conclusions and future work for this project.

MILESTONES	START	DUE	1-Jun	2-Jun	3-Jun	4-Jun	5-Jun	6-Jun	7-Jun	8-Jun	9-Jun	10-Jun	11-Jun	12-Jun	13-Jun	14-Jun	15-Jun
Finalize research aim, objectives, and project scope	1-Jun	1-Jun															
Conduct literature review on collaborative robotics and safety system	1-Jun	30-Jun															
Set up simulation environment and tools (e.g., ROS, Gazebo)	1-Jun	15-Jun															
Begin initial data exploration and 3D vision system setup	20-Jul	17-Jun															
Develop and complete simulations for Task 1 & Task 2 (single and multiple object pick-and-place)	15-Jul	30-Aug															
Start design of robotic system modules: vision, motion planning, manipulation	1-Jul	14-Jul															
Begin implementation of task allocation logic using real-time dataset inputs	10-Jul	30-Jul															
Set up the safety system for the collaborative robot with LIDAR	23-Jul	7-Aug															
Integrate task allocation and safety mechanisms into the robotic system	1-Aug	14-Aug															
Conduct testing for performance evaluation: efficiency, accuracy, collaboration quality	15-Aug	30-Aug															
Analyze test results and write dissertation: methodology, results, and discussion Review and edit full document	15-Aug	30-Aug															
Create presentation slides	18-Aug	28-Aug															
Submit final dissertation and prepare for any presentation or viva requirements	7-Sep	8-Sep															

Figure 1.1: Old Gantt Chart before refining aims and objectives – Part 1

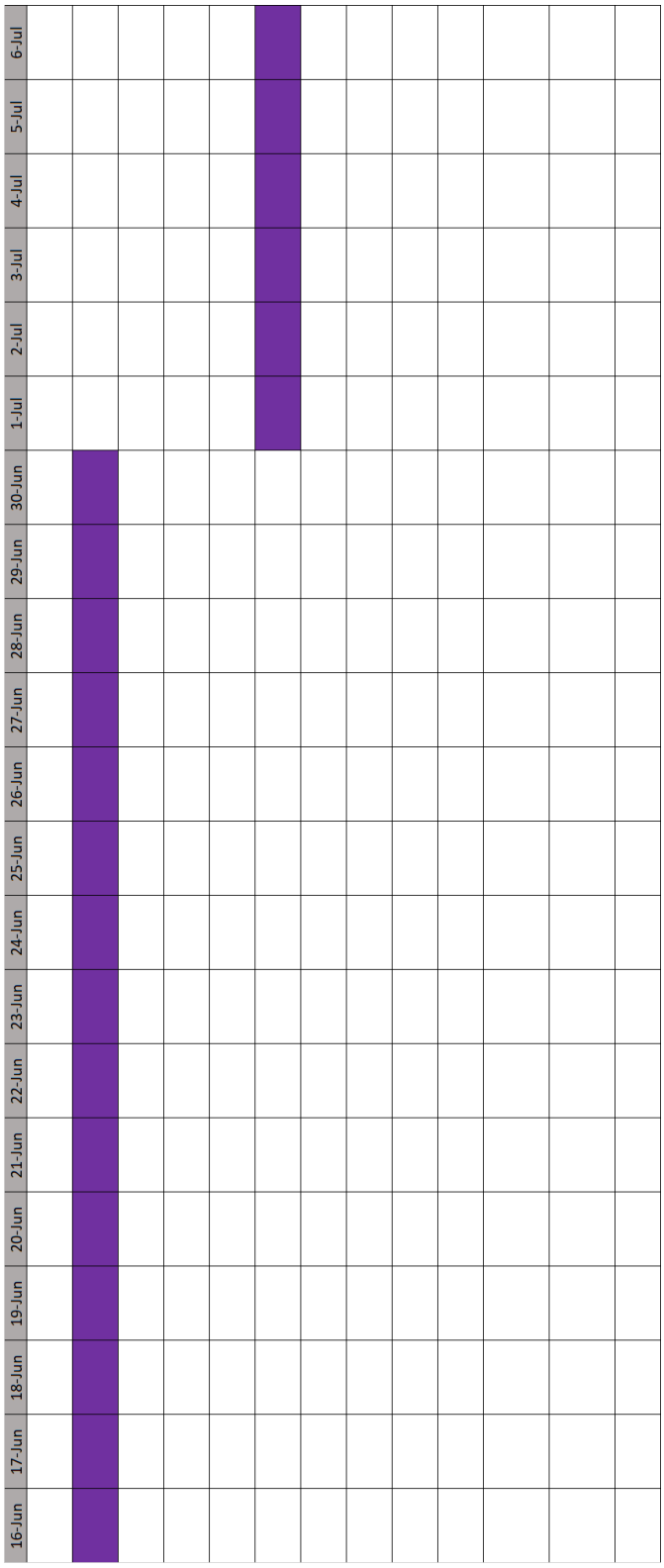

Figure 1.2: Old Gantt Chart before refining aims and objectives – Part 2

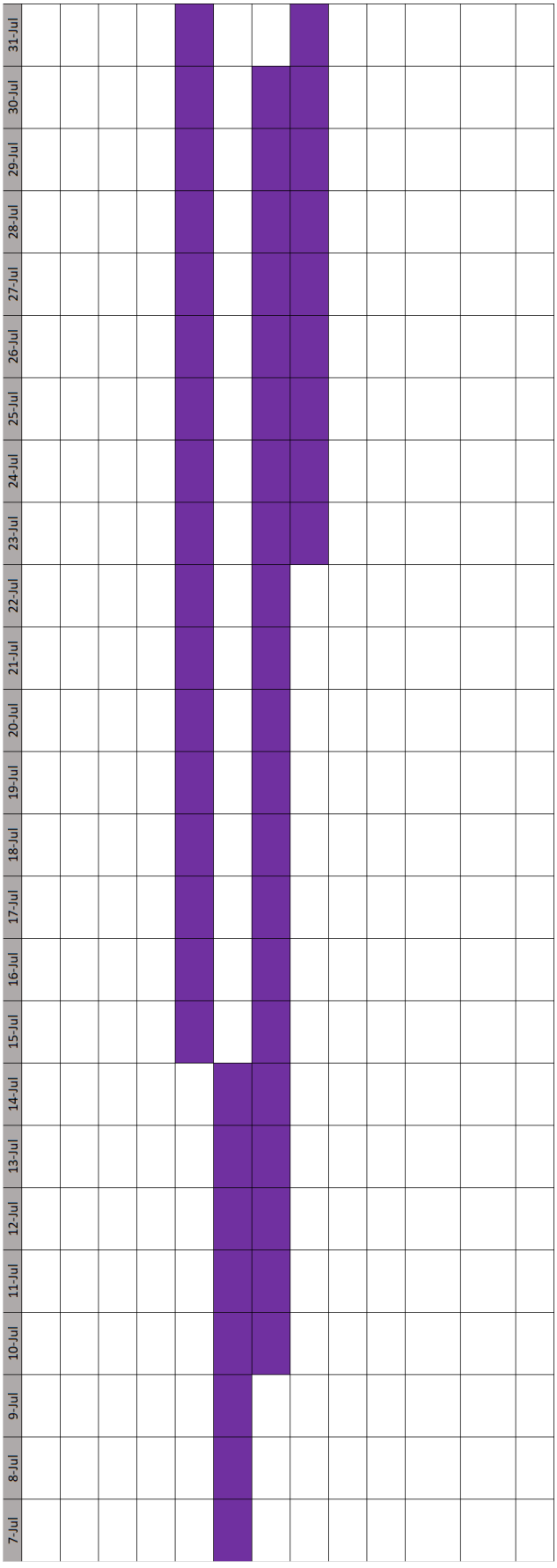

Figure 1.3: Old Gantt Chart before refining aims and objectives – Part 3

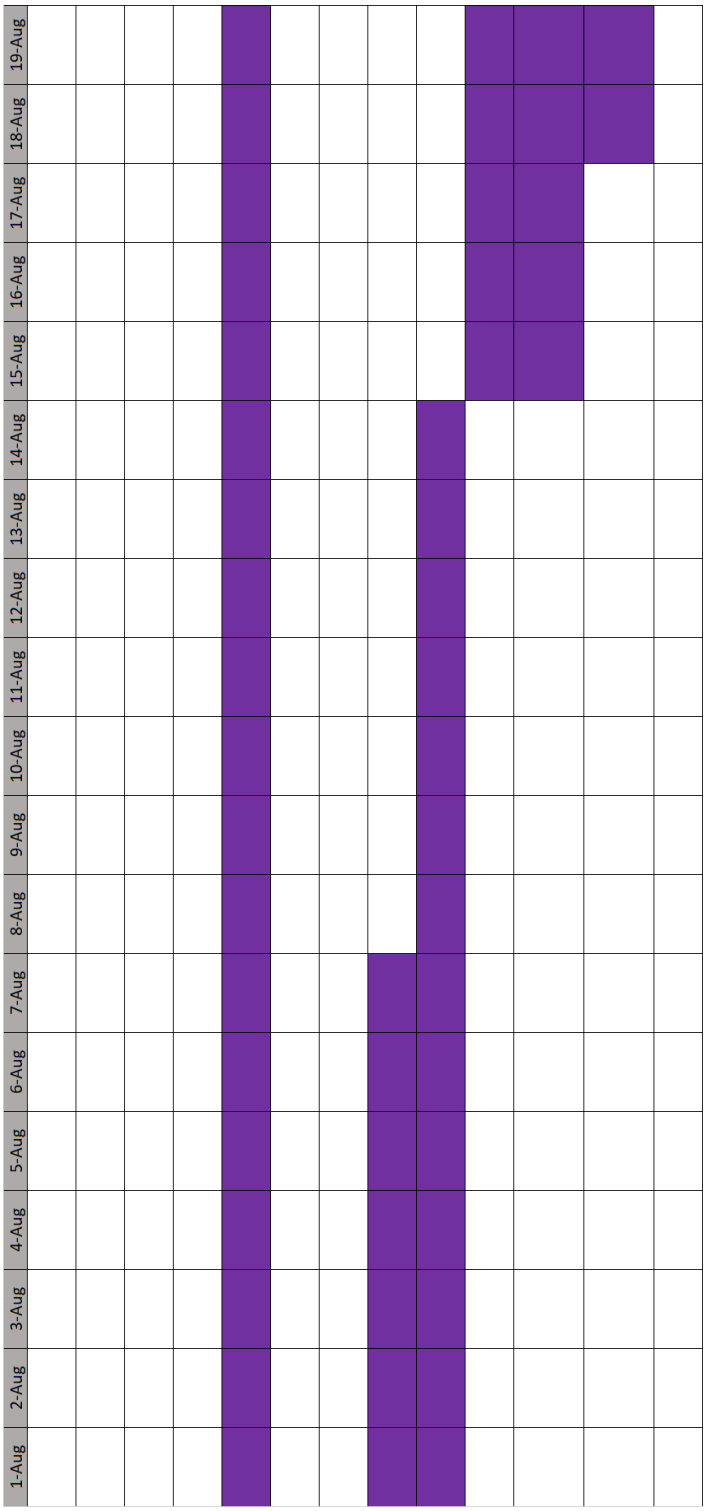

Figure 1.4: Old Gantt Chart before refining aims and objectives – Part 4

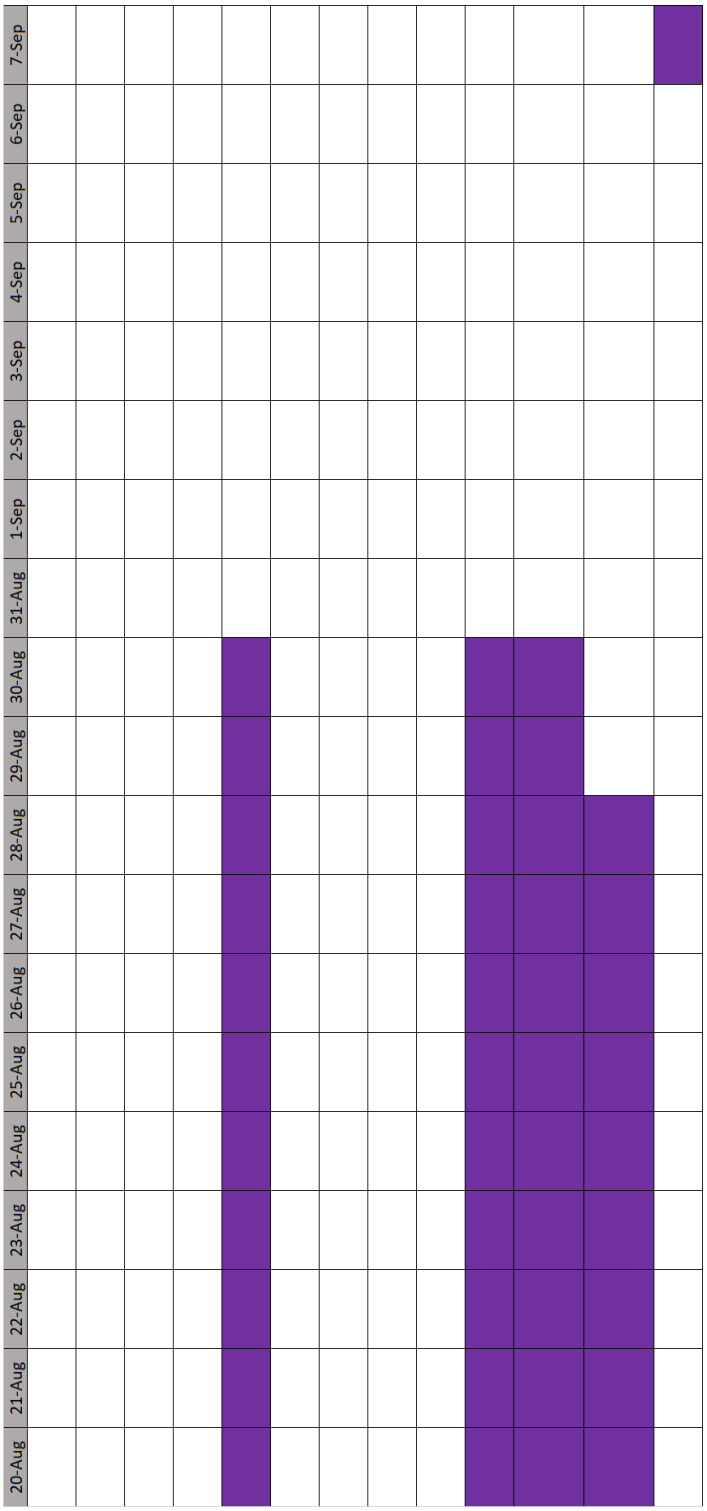

Figure 1.5: Old Gantt Chart before refining aims and objectives – Part 5

Task Name	Duration	Start	Finish
Basic Objectives	**46 days**	**Sun 6/1/25**	**Fri 8/1/25**
Literature review	7 days	Sun 6/1/25	Mon 6/9/25
Set up Simulation Environment and System	17 days	Tue 6/10/25	Wed 7/2/25
Test with different Modules, Sensors, and Motion Planning	10 days	Tue 7/1/25	Mon 7/14/25
Perform Pick-and-Place task with Defined Object	11 days	Thu 7/10/25	Thu 7/24/25
Perform Task 1	11 days	Thu 7/10/25	Thu 7/24/25
Develop a Safety Sytem	3 days	Tue 7/29/25	Thu 7/31/25
Advanced Objectives	**13 days**	**Sat 8/2/25**	**Mon 8/18/25**
Perform Task 2	12 days	Sat 8/2/25	Fri 8/15/25
Integrate assembly tasks with safety	6 days	Sat 8/2/25	Fri 8/8/25
Assess the system safety	3 days	Thu 8/14/25	Mon 8/18/25
Additional Tasks			
Point Cloud Processing	7 days	Thu 7/10/25	Fri 7/18/25
Add Human Models to assess the System	4 days	Sat 8/2/25	Wed 8/6/25
Designing a Hole and a IKEA Table Top	5 days	Thu 7/10/25	Wed 7/16/25
Dissertation Writting	**22 days**	**Sun 8/10/25**	**Sun 9/7/25**
Literature review	7 days	Sun 8/10/25	Mon 8/18/25
Methodology	7 days	Tue 8/19/25	Wed 8/27/25
Conclusion + Introduction	3 days	Thu 8/28/25	Mon 9/1/25
Abstract + Formatting	2 days	Wed 9/3/25	Thu 9/4/25
Preparing for Oral Dissertation	7 days	Mon 8/18/25	Tue 8/26/25
Actual			
Planned			

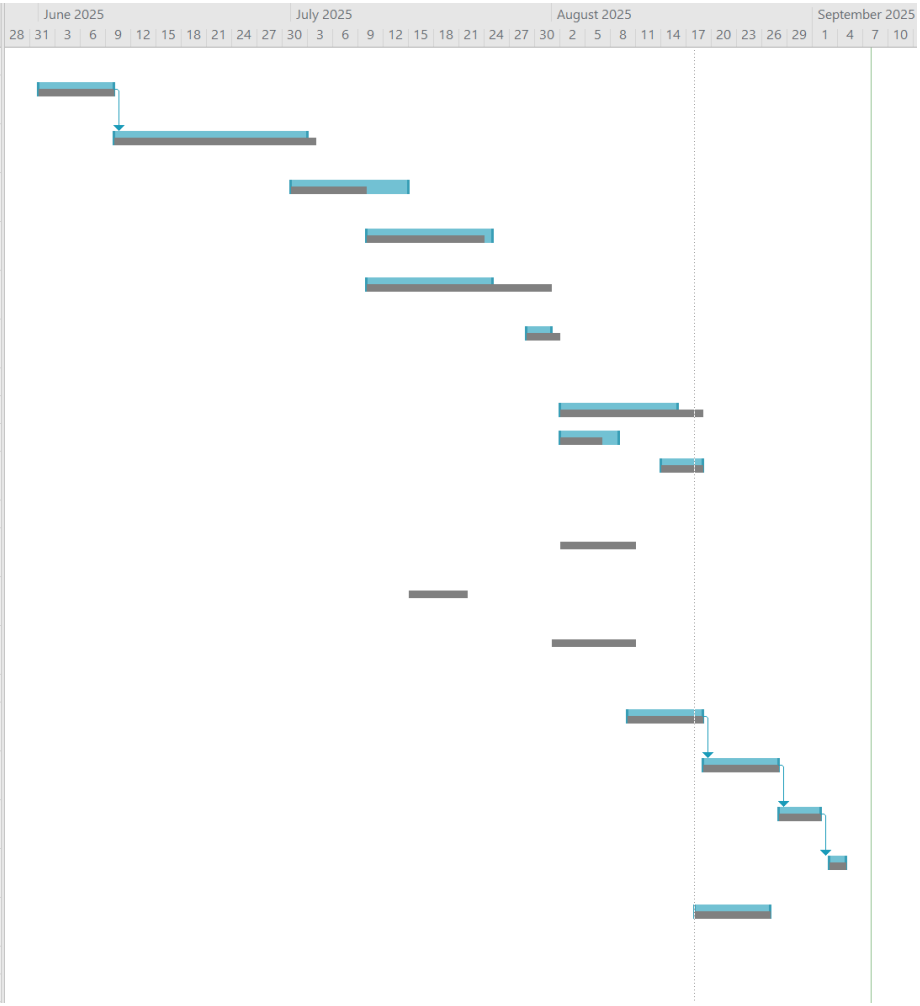

Figure 1.6: New Gantt Chart after refining aims and objectives

Chapter 2

Literature Review

Human-robot collaboration (HRC) is transforming industries such as manufacturing, assembly, and logistics by combining robotic precision with human adaptability. In furniture manufacturing, HRC offers significant potential to automate repetitive tasks, such as pick-and-place and part alignment, while leveraging human decision-making for complex or unstructured processes [10]. However, the field faces challenges in ensuring safety, accuracy, and efficiency in dynamic, human-shared environments. This review examines HRC applications, safety mechanisms, perception and sensing, and motion planning, with a focus on furniture assembly. It identifies research gaps relevant to the proposed system, which utilizes a UR5 manipulator, attaching a gripper, a depth camera, and LiDAR-based safety monitoring.

2.1 Human-Robot Collaboration in Furniture Industry

Furniture manufacturing is one of the potential fields for the use of cobots. The reason is that even industry is keen on automating processes, but technical and scientific knowledge is what they lack, especially in wood product services [9]. HRC has been extensively studied in the automotive, electronics, and logistics industries, where robots handle repetitive or heavy tasks, and humans contribute dexterity and problem-solving [11]. Furniture assembly, however, remains underexplored, despite its suitability for HRC [9].

Recent advancements in collaborative robotics have explored diverse strategies to enhance task coordination, perception, and safety in shared human–robot environments. Rizwan et al. developed a collaborative system using the Baxter robot to assist human operators during table assembly tasks, focusing on decision-making and communication between humans and

robots [15]. Similarly, Darvish et al. employed Baxter in various collaborative tasks, including table assembly, with an emphasis on autonomous decision-making capabilities [42]. While both systems demonstrated effective coordination, their performance relied heavily on robust sensing to manage environmental uncertainty and dynamic human behavior.

Complementary studies have investigated gesture-based guidance, impedance control, and haptic feedback to facilitate safe and intuitive human–robot interaction in assembly tasks. Robots such as the KUKA LBR iiwa and UR10e have been programmed to learn from task demonstrations using techniques such as manual guidance of robot trajectories and trajectory encoding. These systems often incorporate impedance control, which models the robot as a mass–spring–damper system to ensure compliant behavior and reduce interaction forces during physical contact with humans [11]. This control strategy allows the robot to adapt its motion profile in response to external forces, making it suitable for tasks requiring precision and safety, such as furniture assembly.

The Kinova Gen3 platform extends this paradigm by integrating gesture recognition through vision-based sensing or wearable interfaces [11]. It adjusts its trajectory in real time based on human input, enabling fluid collaboration without the need for explicit programming. This responsiveness supports dynamic task allocation and enhances user trust in shared workspaces.

Notably, Suarez et al. demonstrated a high level of dexterity in human–robot collaboration using a humanoid robot guided via a haptic device [21]. Their system enabled real-time manipulation of chair components with sub-millimeter precision, leveraging tactile feedback to refine motion control. This approach highlights the potential of multimodal interaction, combining force sensing, haptic input, and visual feedback to achieve nuanced coordination in complex assembly tasks.

Beyond task execution, several studies emphasize ergonomic considerations in collaborative assembly. Brunello et al. reviewed a range of furniture assembly applications from tables to drawers, highlighting the importance of workspace redesign and ergonomic optimization [11]. Ciccarelli et al. developed a collaborative system using the UR10e robot for kitchen manufacturing tasks such as packing, integrating safety and ergonomic principles. Their risk assessment addressed both spatial layout and operational performance, demonstrating how human-centered design can mitigate physical strain while maintaining productivity [9]. This approach aligns with ISO/TS 15066 standards and offers a practical framework for deploying collaborative robots in ergonomically sensitive environments [18].

Together, these studies underscore the multifaceted nature of collaborative robotics, where planning algorithms, multimodal interaction, and spatial awareness converge to support safe and efficient human–robot cooperation in complex tasks such as furniture assembly. However, the furniture industry remains in the early stages of adopting collaborative robotics, with limited integration of ergonomic safety evaluation, real-time perception, and simulation-based validation. This gap presents an opportunity to develop systems that not only automate physical tasks but also prioritize human well-being and operational adaptability in dynamic environments.

2.2 Safety for Human-Robot Collaboration

Many safety systems help to avoid collisions, keep impact forces to an acceptable level in the event of a crash, make robots aware of humans, and ensure that robots can safely perform tasks in changing environments [9].

2.2.1 ISO 10218-1/2 and ISO/TS 15066

Safety is paramount in HRC to prevent collisions and ensure acceptable impact forces in dynamic environments. Standards and perception-based methods guide safe robot operation.

Safety Standards ISO 10218-1/2 provides risk assessment guidelines for industrial robots, while ISO/TS 15066 extends these for collaborative robots, defining four safety modes [16][17]. The evaluation of potential collision should be in real-time controls of the system [22]. According to this document, there are four main collaborative modes to ensure the safety of humans, shown as Figure 2.1 below [18].

First, the safety-rated monitored stop. In this mode, a cobot stops running when a human interacts with it. Thus, to be able to do this, functional modules should be implemented, such as a sensing system to evaluate the presence of humans in or out of the workspace.

Second, the hand-guiding mode is when a human guides the cobot to teach or adjust its motion through hand contact. For this requirement, the robot control must compensate for the driving forces and gravity to achieve the desired goal configurations. Moreover, the force sensors must be carefully calibrated to stay within the allowed limit. For example, in the work mentioned above, an impedance control, which models the robot as a mass-spring-damper system, is used to reduce the power force.

Third, the speed and separation monitoring is a method that determines the motion direction

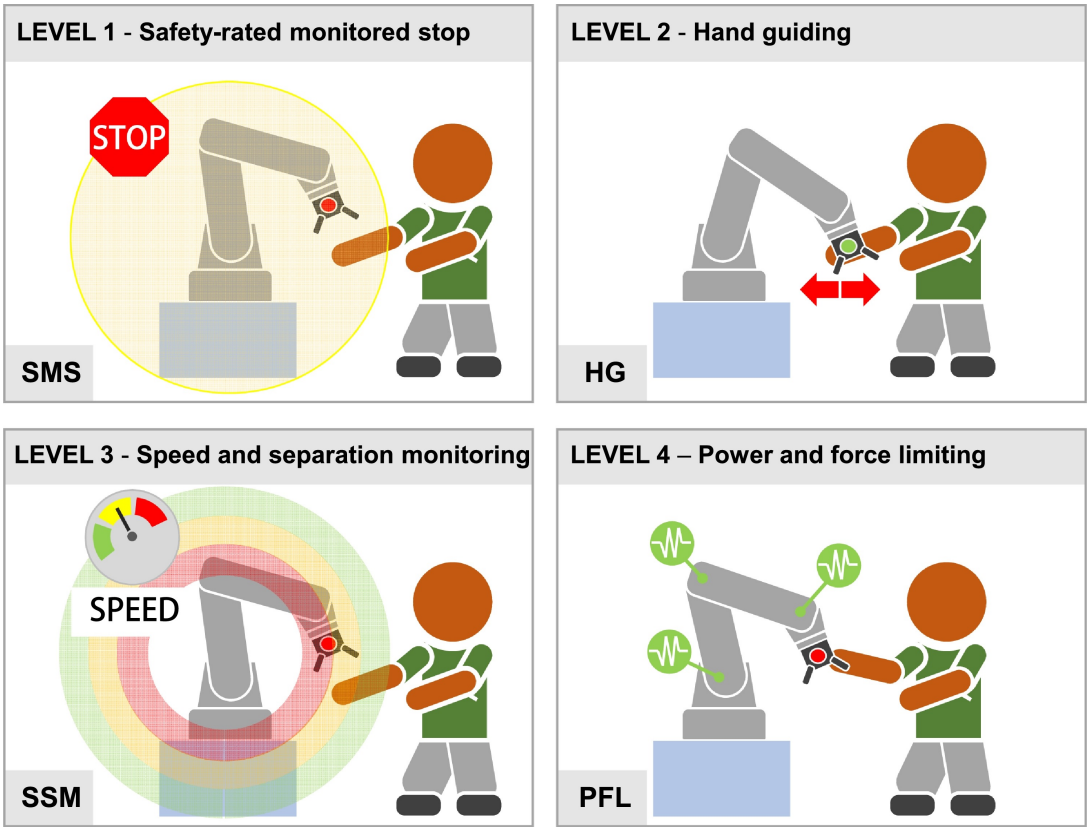

Figure 2.1: Collaborative modes of safety. Source: Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces, and applications [20].

of the robot based on the distance of the humans. For example, the robot operates at full speed when no humans are in the workspace and slows down if a human is close to the system or the yellow zone and fully stops at the red zone or very close to the workspace. To determine the range of the yellow and red zones, or speed, it depends and has to be carefully considered based on the trade-off between productivity and safety.

Lastly, in the power and force limiting, the robot must stop or reduce the power and force under the safety limits when a human is in danger. The robot control has to stop or compensate for the gravity when a collision is detected.

These modes ensure compliance with safety standards, critical for the proposed system's LiDAR-based monitoring of the protective separation distance (PSD). However, the current implementation lacks precise calibration of separation distances and corresponding velocity thresholds, which are essential for optimizing both safety and productivity in dynamic human–robot interaction scenarios [19].

2.2.2 Vision-based perception for safety

Vision-based perception enhances safety by detecting humans in 3D space. According to the research, there are two primary methods for detecting humans: vision-based methods and inertial sensor-based methods, such as special suits designed to capture motions. Some research primarily utilizes a 3D camera or a depth camera to detect humans in three-dimensional spaces, calculating the distance between humans and the workspace [23]. Differently, Ahmad and Plapper employ a top-mounted time-of-flight (TOF) sensor to monitor both human and robotic activity within the shared workspace, aiming to prevent collisions through overhead depth sensing [24]. While this approach offers a broad field of view and simplifies installation, it introduces several limitations that may compromise system reliability. Specifically, the overhead perspective can result in occlusions, particularly when objects or robot arms obstruct the line of sight, leading to inaccurate human detection or misclassification. Such misdetection not only poses safety risks but also disrupts task execution, as the robot may unnecessarily halt or slow down, thereby reducing overall productivity. These challenges underscore the importance of selecting sensor configurations that balance coverage, accuracy, and responsiveness in dynamic collaborative environments.

Additionally, sensor fusion is also used to fuse data from multiple devices, such as cameras and sensors, to improve the reliability in ensuring human safety. Notably, Bwidi et al. (2017) built upon Bwidi's (2014) research on the importance of sensor fusion, utilizing both RGBD cameras and force/torque sensors to analyze human safety at various levels. However, due to many sensors, the computation process is complex and slow, which affects the reaction of the robots [25]. Alternatively, Vogel et al. employed an LED–DLP projector system to visually delineate safe zones around a KUKA robot, effectively separating human and robot operational areas [26]. This approach leverages spatial light projection to create dynamic visual boundaries, enhancing human awareness and reducing the likelihood of accidental encroachment. While it does not directly influence the robot's control logic, it serves as a proactive safety measure by guiding human behavior through intuitive visual cues, particularly useful in environments where physical barriers are impractical or undesirable.

In addition, Cogurcu et al. proposed a method for the visualization of a safety zone for a robot arm. To support the study, they employed virtual reality (VR) and augmented reality (AR) to visualize a non-physical world in real life by using Microsoft HoloLens 2 [41]. The safety zone is divided into four main categories: small cuboids, small cylinders, large cuboids with

cage bars, and large cylinders with cage bars. Participants were then instructed to collaborate with the robot in a basic pick-and-place task within these zones and give feedback. This is an interesting idea, which opens a new way for the safety of collaborative robots, but it takes time to validate and test in more complex environments and task scenarios.

To conclude, vision-enabled methods played a crucial part in perception-based safety systems. Real-time modeling of the operational environment using precise RGB-D and LiDAR-based sensors enabled rapid robot planning, obstacle evasion, and comprehension of human presence, resulting in more anticipatory and secure robotic behaviors. The proposed system leverages LiDAR for real-time human detection and depth camera data for precise object localization, mitigating computational delays through optimized Point Cloud Library (PCL)-based processing. This design choice is directly informed by the latency challenges observed in prior work; for instance, Bwidi et al.'s report of slowed robot reaction times due to multi-sensor computation [25] and Vogel et al.'s implicit trade-off, where projector-based visual guidance, while enhancing safety awareness, does not integrate with real-time control loops [26]. Therefore, the complexity and computational intensity of sensor fusion and projector-based processing can introduce latency, potentially affecting task efficiency despite ensuring optimal safety compliance.

2.3 Robot Capabilities in Assembly Tasks

Effective human–robot collaboration (HRC) in semi-structured environments, such as furniture assembly, requires a combination of robust perception and intelligent motion planning. These capabilities enable the robot to detect and interpret objects, understand spatial relationships, and execute tasks with precision while maintaining safety and adaptability in shared workspaces.

2.3.1 Perception and Sensing

Perception forms the foundation of autonomous decision-making in collaborative robotics. Robots rely on either 2D or 3D sensing modalities to interpret their environment. Traditional 2D vision systems, typically based on monocular cameras, are suitable for planar workspaces and simple tasks [27]. However, they are inherently limited in terms of depth perception and spatial reasoning, making them inadequate for complex manipulation tasks that involve variable

geometries or occlusions. Moreover, 2D systems often require fixed gripper configurations and predefined lighting conditions, which restrict their flexibility in dynamic environments.

To overcome these limitations, modern collaborative systems increasingly adopt 3D sensing technologies. RGB-D cameras such as the Microsoft Kinect and Intel RealSense D435 have become widely used due to their ability to capture both color and depth information simultaneously. These sensors generate depth images that can be converted into 3D point clouds using the camera's intrinsic parameters, enabling the spatial reconstruction of the environment [27]. This capability is particularly valuable in furniture assembly, where components such as cylindrical legs and mounting holes must be accurately identified and manipulated.

Guiguang et al. also demonstrated that depth-based perception allows for reliable detection of objects [27]. In the proposed system, this principle is applied through a Point Cloud Library (PCL)-based pipeline, which processes depth data to extract geometric features relevant to furniture components. The pipeline supports real-time segmentation and classification, enabling the robot to distinguish between parts, estimate their poses, and plan appropriate grasping or insertion actions.

Furthermore, the integration of depth sensing with LiDAR-based human detection enhances situational awareness [27]. While the depth camera focuses on object-level perception, LiDAR continuously monitors human presence in the workspace, supporting the enforcement of dynamic protective separation distances (PSDs). This dual-sensor approach strikes a balance between precision and safety, enabling the robot to operate efficiently while adapting to human proximity.

2.3.2 Motion Planning

Motion planning ensures collision-free, efficient trajectories. According to the papers, the author proposed different methods for the existing path planning algorithm and an advanced algorithm built on that [28]. Probabilistic Roadmaps (PRM) and its variant PRM* sample configuration spaces to build reusable roadmaps, suitable for multi-query tasks but less efficient for single-query scenarios. PRM* achieves asymptotic optimality but requires significant computation. However, PRM is a multi-query application, reusing the map and avoiding building a new map every time it runs. Therefore, this algorithm is used for repeated queries rather than a one-off query [28].

The Rapidly-exploring Random tree (RRT), however, is a single query application; it can

build a tree from the root with feasible trajectories and create a path connecting all points that are near the tree [28]. Similarly, Bidirectional RRT (BiRRT) employs the same logic and constructs a path plan using two trees. It is particularly effective for planning motions where the robot's end effector must satisfy specific pose constraints, such as the same orientation or surface. Building on this, the Constrained Bidirectional Rapidly The Constrained Bidirectional Rapidly-exploring Random Tree (CBiRRT2) algorithm adds task-space constraints to the random sampling process, thereby increasing the efficiency and feasibility of motion planning in complex environments [30]. By enforcing pose or surface constraints during tree expansion, CBiRRT2 enables more accurate trajectory generation for manipulation tasks such as furniture assembly. However, its performance can degrade in highly cluttered or dynamically changing environments, where constraint satisfaction may lead to excessive sampling rejection or prolonged planning times, limiting its applicability in real-time collaborative scenarios. Ultimately, the RRT* or Optimal RRT calculates the cost of the path to find the shortest plan, but it takes time and is computationally intensive for each path planning.

For the collaborative robot, Rizwan et al. present a hybrid conditioning planning, which generates robust plans in environments with incomplete knowledge or partial observability [15]. It extends traditional conditional planning by integrating external feasibility checks during the planning phase to ensure that all actions in the plan are physically executable. While Rizwan et al.'s hybrid planning handles partial observability, its reliance on answer set programming may limit real-time performance in dynamic furniture assembly tasks and decrease productivity. Additionally, Stevo et al. introduce a genetic algorithm to optimize robotic arm trajectories by encoding motion paths as candidate solutions and evolving them through selection and mutation [29]. This approach enables efficient navigation in complex environments, balancing objectives like obstacle avoidance, smoothness, and energy use.

To conclude, the proposed system should be evaluated across multiple motion planning algorithms to explicitly determine which approach best suits the requirements of collaborative furniture assembly tasks. Such a comparative analysis would help balance safety, efficiency, and responsiveness in real-world deployment.

2.4 Conclusion and Research Gaps

The reviewed literature demonstrates substantial progress in human–robot collaboration (HRC), particularly in the automotive and electronics industries. These studies highlight the integration of planning algorithms, multimodal interaction, and spatial awareness to support safe and efficient cooperation. However, several critical gaps remain in the context of collaborative furniture assembly, which is characterized by semi-structured environments, irregular geometries, and ergonomic constraints.

- While HRC is well established in structured domains, its application to furniture assembly is underexplored [9]. The irregular shapes, varied materials, and unstructured layouts typical of furniture tasks pose unique challenges for perception, manipulation, and safety.
- ISO/TS 15066 provides foundational safety guidelines, including speed and separation monitoring [18]. However, there is a lack of integrated frameworks that optimize these parameters without compromising task efficiency, especially in dynamic human–robot shared workspaces [19].
- Multisensor systems, such as LiDAR and RGB-D cameras, enhance situational awareness and safety. Yet, their fusion introduces computational overhead, which can delay real-time decision-making and reduce responsiveness in collaborative scenarios [25].
- Numerous motion planning algorithms (e.g., PRM, RRT, BiRRT, CBiRRT2) have been proposed [28]. However, selecting an optimal approach that balances safety, adaptability, and computational efficiency for semi-structured tasks remains unresolved. These tasks demand robust perception, constraint-aware planning, and responsiveness to human presence and environmental variability.

The proposed system addresses these gaps by developing tailored depth camera pipelines for cylinder and hole detection, suited to furniture parts. Integrating LiDAR-based Speed and Separation Monitoring (SSM) with RRT-based re-planning for safe, and efficient HRC. Optimizing PCL processing to minimize computational delays, enabling real-time responsiveness. Applying RRT variants in the MoveIt framework for dynamic, high-degree-of-freedom (DoF) planning in semi-structured environments.

In summary, although collaborative robotics shows great potential for improving safety, efficiency, and ergonomics in furniture assembly, the field is still in its early stages of develop-

ment. Future research should focus on developing adaptive, safe, and efficient systems tailored to the unique challenges of the furniture industry, thereby bridging the gap between industrial interests and technical feasibility.

Chapter 3

Methodology

This chapter presents the designed system for a human-collaborative system, including both hardware and software. Section 3.1 mentions the system overview of the outline system, including design and workflow. Section 3.2 is followed by a detailed description of each hardware component, sensing method, simulator, and object used in the simulation world. Section 3.3 introduces a path planning plan and its application for the system. Section 3.4 presents safety protocols and operation modes of the system. Lastly, Section 3.5 focuses on risk assessment. Along with each section, there are figures and pictures to illustrate the problems.

3.1 System Overview

3.1.1 System Design

The proposed system establishes a human-robot collaborative environment for furniture assembly, with a particular focus on integrating safety mechanisms that comply with ISO 10218-1/2 and ISO/TS 15066 standards [16] [17] [18]. The system is designed by dividing it into two zones with high and low levels of risk to allow the robot and the human operator to share the same workspace, where tasks are distributed according to their respective strengths. The robot executes repetitive, heavy, and ergonomically demanding operations with high accuracy, while the human provides adaptability, decision-making, and fine manipulation skills.

A key feature of the system is the integration of a multi-layered safety architecture, which ensures that collaboration remains productive while prioritizing worker protection. The system is equipped with perception and sensing technologies, such as LiDAR, that continuously mon-

itor human presence. Real-time safety protocols dynamically adjust the robot's motion. For example, slowing down, pausing, casually working, or rerouting trajectories when the human enters predefined zones. This is supported by path and task planning algorithms, which optimize both efficiency and safety during operation. The system design is organized into five core components:

- Robot Arm and End Effector, which is responsible for executing assembly operations.
- Perception and Sensing, which monitors the environment and human activity.
- Path and task planning, which generates safe and efficient trajectories.
- Safety Protocols and Operating Modes, which implement collaborative modes such as speed and separation monitoring, and power and force limiting.
- Risk Analysis, which evaluates compliance with safety standards and assesses residual risks in the collaborative process.

Together, these components create a human-centered collaborative assembly system that addresses ergonomic challenges in the furniture industry, improves productivity, and ensures compliance with international safety requirements.

3.1.2 Workflow of the System

The collaboration process begins with raw furniture components randomly arranged in the workspace. The robot arm, equipped with a Robotiq 2F-85 end effector, performs lifting, aligning, and assembly operations, utilizing a depth camera with point cloud processing. The human operator intervenes as required, particularly for tasks that require flexibility or to supply and collect raw and final parts. Figure 3.1 shows the workflow of the system, depending on the sensor detection.

The workflow begins with an essential initial step involving point cloud processing, a technique that uses sensor data to analyze the three-dimensional structure of objects. During this phase, the robotic system scans all objects within its workspace, such as cylinders or chair components, to determine their central points, known as centroids. These centroids are then compiled into a comprehensive list, which includes both cylinder centroids and hole centroids, serving as a road map for the robot's subsequent actions. This list is crucial because it allows

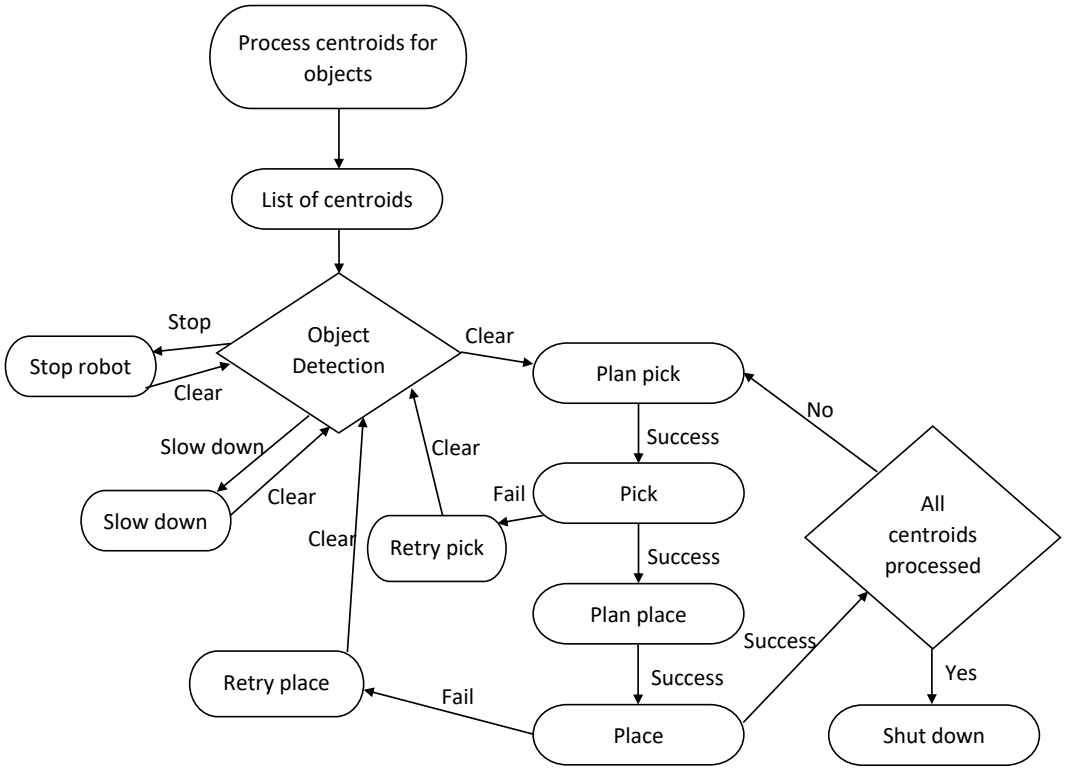

Figure 3.1: This flowchart illustrates the system workflow designed to simultaneously enhance safety and optimize productivity. The process begins with centroid extraction, followed by collision checking, and culminates in task execution.

the system to manage multiple items in a single operation, enhancing efficiency in tasks like assembling furniture pieces. However, this step can be affected by challenges such as sensor noise or inaccurate detection, which may lead to minor positional errors.

Following the creation of the centroid list, the workflow progresses to the object detection system, a critical safety checkpoint. This system employs sensors, likely including LiDAR, to identify any obstacles or objects present in the designated zones—namely, the front zone for raw material supply and the left zone for finished product collection. The detection process triggers different responses based on the location and nature of the obstacles, ensuring compliance with safety standards like ISO/TS 15066. If an obstacle is detected in the left zone, where human presence is less critical, the system initiates a “slow down” command. This reduces

the robot's speed to maintain a safe distance from the human, allowing tasks to continue with minimal disruption. In contrast, if obstacles are detected in the front zone or both zones simultaneously, scenarios posing higher collision risks, the system immediately activates a "stop robot" command. This halts all operations instantly to prevent potential accidents, prioritizing human safety over productivity. These adaptive responses demonstrate the system's ability to dynamically adjust to real-time conditions, though they may introduce delays that impact workflow efficiency.

Once the path is confirmed as clear, the workflow moves to the core operational phase, focusing on planning and executing pick-and-place tasks. This process begins with "plan pick", where the robot calculates an optimal path to approach and grasp the object at its centroid, likely using motion planning algorithms such as RRT from the Open Motion Planning Library (OMPL). If the planning is successful, the robot proceeds to the "pick" action, attempting to grasp the object. However, if the pick fails, likely due to grip issues or misalignment, or if the robot enters stop mode due to a sudden obstacle detection, the system triggers a recovery mechanism. This involves replanning the execution, allowing the robot to retry the pick action. This retry capability enhances reliability by addressing common failures without requiring manual intervention, though repeated attempts could extend task duration.

After a successful pick, the workflow advances to "plan place" where the robot designs a path to position the object, such as inserting a cylinder into a hole. Upon successful planning, the "place" action is executed. If placement fails, the system again initiates a recovery process, replanning the placement action. This fault-tolerant design ensures that the robot can correct errors, such as misalignment caused by RANSAC Circle 2D sensitivity, and complete the task. The iterative nature of this loop reflects a commitment to precision, though it may introduce minor delays, highlighting a trade-off between accuracy and speed.

The workflow concludes with a decision point labeled "all centroids processed". This step checks whether every centroid in the list has been processed. If the answer is "no" the system loops back to the object detection phase or the next centroid in the list, repeating the pick-and-place cycle until all objects are processed. This looping mechanism supports high-volume operations, such as assembling multiple chair legs, and demonstrates the system's capacity for continuous task execution. Once all centroids are processed successfully and in the correct order, the workflow ends with a "shutdown" command, safely powering off the robot to conserve energy and avoid risks in an idle state. Alternatively, if further tasks are required, the system

can restart the cycle, ensuring flexibility for ongoing operations.

3.2 Hardware and Simulation Setup

This section describes virtual environments used in the investigation. The experimental platform includes an industrial robotic manipulator, end-effector, and multi-modal sensor system with simulation tools to test safety protocols and collaborative performance. Simulation ensures repeatability, scalability, and safe verification of safety systems before moving toward real-life systems.

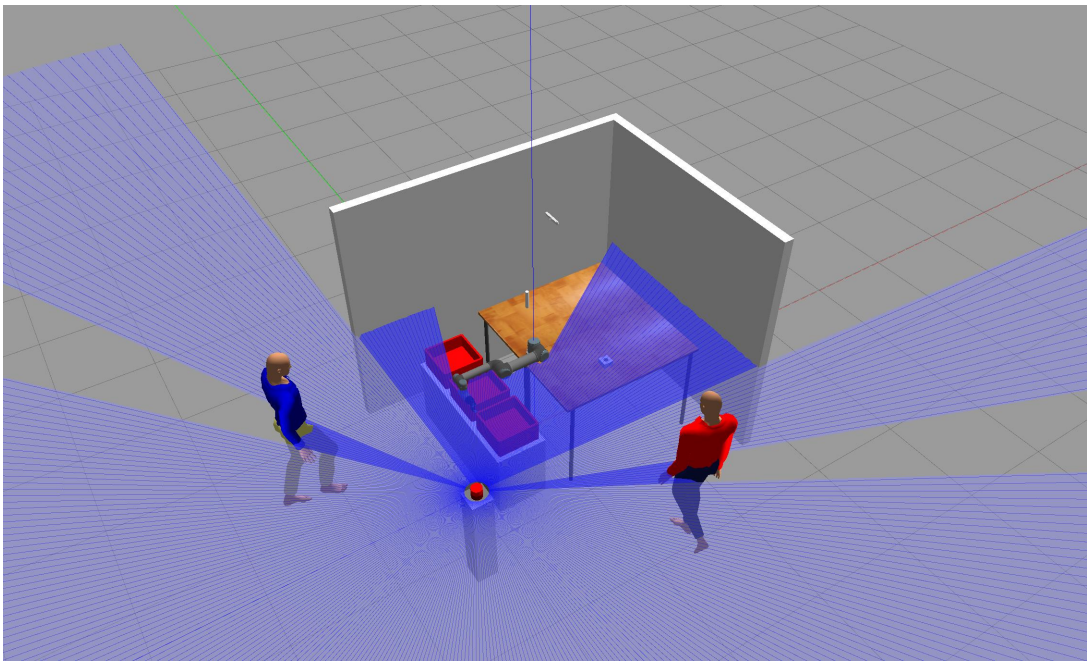

Figure 3.2: The Gazebo simulation setup for Task 1 incorporates human models to evaluate the system's safety procedures under realistic interaction scenarios. In this task, the workspace consists of a single cylindrical object and one corresponding hole on the table, representing a simplified assembly environment for initial validation.

3.2.1 ROS and Gazebo Integration

The software architecture of the human-robot collaboration (HRC) system leverages the Robot Operating System (ROS) to provide a modular, flexible middleware framework for integrating hardware, perception, and control components [35]. ROS facilitates seamless communication between the UR5 manipulator, depth camera, LiDAR sensor, and motion plan-

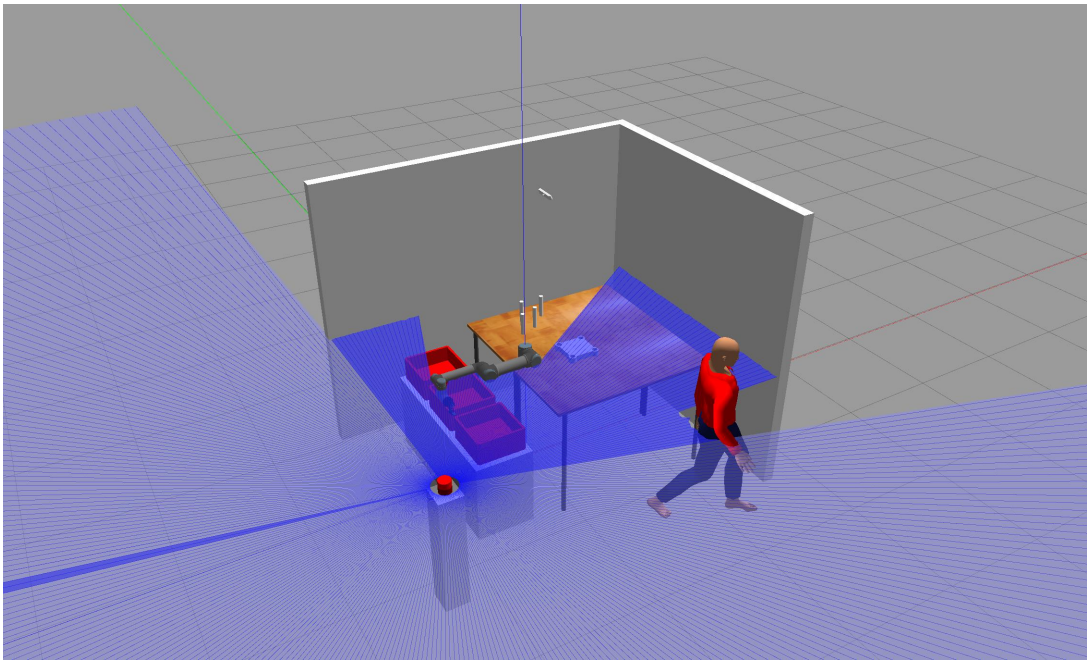

Figure 3.3: The Gazebo simulation setup for Task 2 incorporates human models to evaluate the system's safety procedures in a more complex assembly scenario. This task involves four cylindrical components and a chair top featuring four corresponding holes, inspired by the design of the IKEA UTTER chair. The setup aims to replicate realistic human–robot interaction during multi-part assembly, enabling validation of collision avoidance and ergonomic safety protocols.

ning modules, ensuring robust data exchange in dynamic furniture assembly tasks. Gazebo, a physics-based robotics simulator, enables the realistic testing of systems in a controlled virtual environment before hardware deployment [36]. Together, ROS and Gazebo support iterative development, enabling the evaluation of perception pipelines, motion planning, and safety protocols that are directly transferable to the physical platform. Figures 3.2 and 3.3 describe the system setup in Gazebo with different tasks.

The MoveIt framework, integrated with ROS, handles path planning, collision detection, and trajectory optimization [38]. It uses the Open Motion Planning Library (OMPL) to implement Rapidly-exploring Random Tree (RRT) algorithms (RRT, RRTConnect, RRT*), generating collision-free trajectories for pick-and-place tasks.

3.2.2 Objects

The test objects in this study are specifically designed to mimic the common handling and assembly challenges that arise when building furniture. In the field of robotics benchmarking, researchers often rely on standard geometric shapes for precise and repeatable testing, including simple forms like cubes and cylinders, as highlighted in studies [37]. These basic shapes enable controlled experiments. This allows variables to be easily managed and assesses a robot's accuracy, grip strength, and movement efficiency with irregular shapes in an unstructured environment, solving the problem in the research gap.

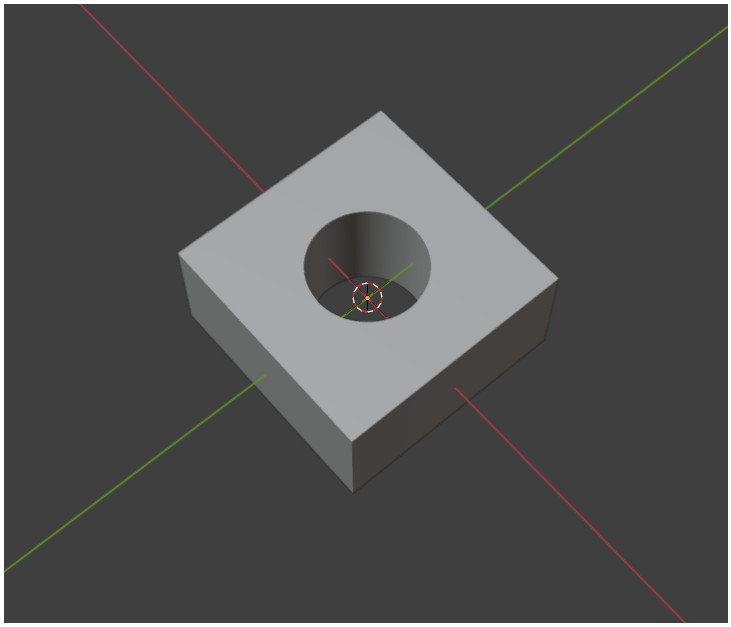

Figure 3.4: A custom-designed hole was created using Blender for insertion testing in Task 1. This component serves as a simplified target geometry for evaluating the robot's precision and alignment capabilities during the simulated assembly process.

To make the experiments more relevant to actual furniture assembly, custom-made objects are created for pick-and-place operations, which involve picking up items and positioning them accurately. For instance, in Task 1 shown in Figure 3.4, a straightforward cylinder (1.2 m height, 0.2 m radius) is used alongside a specially crafted hole, designed in Blender, and its setup is shown in Figure 3.3. This setup simulates the insertion of parts, such as fitting a cylinder into a pre-drilled hole (0.23 m radius, 0.25 m height), testing the robot's precision in alignment and force application to avoid damage. Moving to Task 2, shown in Figure 3.5, the system has developed a more complex custom chair top (1.4x1.4 m, 0.25 m height, and 0.23 m radius per hole) and setup, as shown in Figure 3.3. This design, also created in Blender, draws

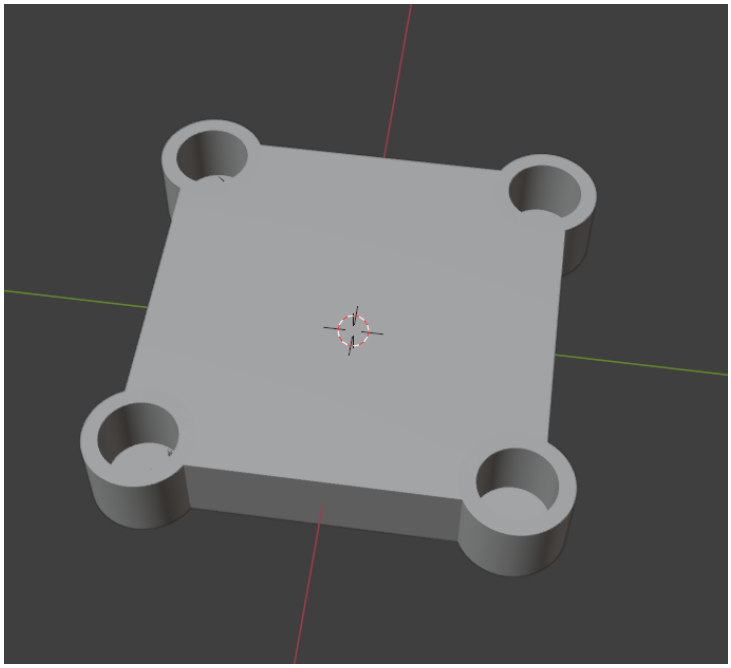

Figure 3.5: The tabletop used in Task 2 was custom-designed in Blender, drawing inspiration from the IKEA UTTER chair. This component features four predefined holes to accommodate cylindrical legs, enabling realistic simulation of multi-part furniture assembly and facilitating safety validation in human–robot collaborative scenarios.

inspiration from the simple design of IKEA's UTTER children's chair. This piece is virtually assembled with four cylindrical legs, representing the process of attaching supports to a seat base. By incorporating these elements, the task assesses the robot's ability to handle irregular shapes, maintain stability during placement, and adapt to multi-step assembly sequences, which are crucial for efficient furniture production in real-world manufacturing settings.

3.2.3 Robot Arm and End Effector

Among the product family, UR3, UR5, and UR10 are widely recognized within the robotics industry and research community [31]. The manipulator selected for this study is the UR5, a 6-degree-of-freedom (6-DOF) industrial robot commonly used in both research and industrial applications. Its flexible, modular integration and comprehensive ROS (Robot Operating System) support make it well-suited for developing and verifying human–robot collaboration (HRC) methods.

For grasping, the Robotiq 2F-85 parallel-jaw gripper is employed due to its compatibility with UR5. Guiguang et al. proposed that a parallel gripper is commonly used in pick-and-place

tasks [27]. Additionally, literature demonstrates that parallel-jaw grippers offer a robust and reliable solution for structured pick-and-place tasks, such as furniture assembly, where precision and repeatability are prioritized over dexterous manipulation [32]. While anthropomorphic or multi-fingered grippers offer higher dexterity, they introduce additional control complexity and are more suitable for highly complex tasks [27]. Thus, this study prioritizes robustness and reliability over dexterity to ensure consistent task execution.

3.2.4 Perception and Sensing

This section introduces the key sensing modalities, LIDAR and depth cameras, that underpin the perception capabilities of the system. LIDAR enables precise spatial mapping and obstacle detection, essential for safety monitoring and collision avoidance. Depth cameras complement this by providing rich point cloud data for object recognition and pose estimation, which are critical for accurate manipulation in furniture assembly tasks. Together, these sensors enhance situational awareness and support robust human–robot collaboration in dynamic environments.

3.2.4.1 The Use of LiDAR

In HRC applications, LiDAR sensors are commonly employed to continuously monitor the robot's workspace for unexpected intrusions, such as objects or human presence, enabling real-time hazard detection and response [34]. This technology leverages its wide field of view (FOV), reliable distance measurements that are both fast and consistent, and robustness to environmental variations, making it a well-established solution in safety-critical HRC scenarios. For instance, unlike vision-only systems, which can be unstable under inconsistent lighting conditions or occlusions, LiDAR operates effectively in diverse illumination settings, providing precise 3D or 2D spatial data without reliance on external light sources [33]. Additionally, LiDAR systems are often more lightweight and computationally efficient compared to complex sensor fusion approaches that integrate multiple modalities, such as cameras, radar, and ultrasonics, reducing overall system complexity while maintaining high reliability for tasks like speed and separation monitoring (SSM), avoiding the heavy computation of sensor fusions in the research gap.

In HRC safety protocols, LiDAR facilitates the implementation of dynamic protective separation distances (PSD), where the robot's speed is adjusted or halted based on detected proximity to humans or obstacles [33]. This is particularly valuable in industrial settings, such

as assembly lines, where humans and robots share workspaces to enhance productivity and flexibility [34]. Common configurations include 2D LiDAR for planar scanning (floor-level obstacle detection) or 3D LiDAR for volumetric coverage, with the latter offering advantages in detecting elevated intrusions like human arms or objects [33]. However, full-FOV LiDAR can introduce computational overhead due to processing large point clouds, potentially leading to latency in real-time applications.

To address these challenges in the context of this project's task, the LiDAR's scanning range is intentionally reduced to a narrow beam of approximately 20 degrees, effectively functioning as a focused laser rangefinder rather than a wide-area scanner. This configuration prioritizes targeted monitoring of high-risk zones, such as the robot's immediate operational path, while minimizing data volume and processing demands, making it ideal for resource-constrained or embedded systems in HRC. Specifically, distance data from rays 1 to 20 degrees set up on the front of the zone are averaged to represent the "front zone", providing a robust estimate of forward clearance that accounts for minor angular variations or noise. Similarly, rays 92 to 109 degrees are averaged for the "left zone", enabling side-specific detection, such as for lateral human approaches. These zonal averages can be computed using simple statistical methods, such as mean filtering over the selected angular sectors, to derive minimum distances for PSD calculations.

Suppose any zonal average falls below a predefined safety threshold, based on ISO/TS 15066 standards for collaborative robots. In that case, the system triggers a response, such as reducing robot velocity or initiating an emergency stop. This zoned approach enhances efficiency by focusing computational resources on critical areas, potentially integrating with broader HRC frameworks, such as predictive motion planning. Limitations include potential blind spots outside the narrow FOV, which could be mitigated through multi-sensor setups or robot-mounted LiDAR for adaptive scanning. Experimental validation in this study could involve benchmarking against full-FOV systems in simulated HRC scenarios, measuring metrics like detection latency, false positives, and energy consumption to demonstrate the trade-offs and benefits of this reduced-range design [33].

3.2.4.2 The Use of Depth Camera (Point Cloud Processing)

Introduction

In addition to the LiDAR-based monitoring system, a depth camera is integrated into the human-robot collaboration (HRC) setup to provide dense 3D spatial information for enhanced object detection, pose estimation, and scene reconstruction. Depth cameras, such as the Microsoft Kinect or Intel RealSense, generate high-resolution point clouds that capture the geometry of objects within the robot's workspace, enabling precise localization and manipulation of custom-designed furniture parts. Unlike 2D vision systems, depth cameras offer robust performance under varying lighting conditions and provide critical depth data for estimating the centroid position and orientation of objects relative to the robot's base coordinate system [27]. This capability is essential for tasks such as robotic grasping, assembly, and precise placement of furniture components in a shared HRC environment.

Camera Setup

The depth camera, an RGB-D sensor (Microsoft Kinect), is mounted on a fixed rig above the robot's workspace at a height of 1.2 meters to ensure a reliable view of the furniture parts and surfaces. Operating at 30 Hz, the camera captures RGB data with a field of view (FOV) that encompasses the entire workspace. Depth data is published as a sensor_msgs/ PointCloud2 message via ROS-Noetic on the topic /rgbd_camera_depth/depth/points. The camera's frame (rgbd_camera_depth_optical_frame) is calibrated to the robot's world frame using a transformation matrix obtained via the tf2_ros transform listener, ensuring accurate mapping of point cloud coordinates for robotic manipulation.

Object Detection Pipelines

Two distinct processing pipelines, in addition to a preprocessing step, are implemented in C++ using the Point Cloud Library (PCL) within a ROS environment to handle the detection of cylindrical objects and circular holes. Both pipelines process raw point clouds to isolate relevant features and compute their centroids for tasks such as grasping (cylinders) or alignment (holes). The pipelines share initial preprocessing steps but diverge in segmentation to address the unique geometric characteristics of cylinders and holes.

First, regarding common preprocessing, raw point clouds are stored as /sensor_msgs/ PointCloud2 messages, converted to PCL PointXYZRGB format, and saved as a PCD file in a directory for debugging. A flag *has_saved* prevents redundant saves, enabling the system to save once required. A voxel grid filter downsamples the point cloud to reduce computational load. For cylinder detections, a leaf size of 0.001 m is used; for hole detection, a slightly

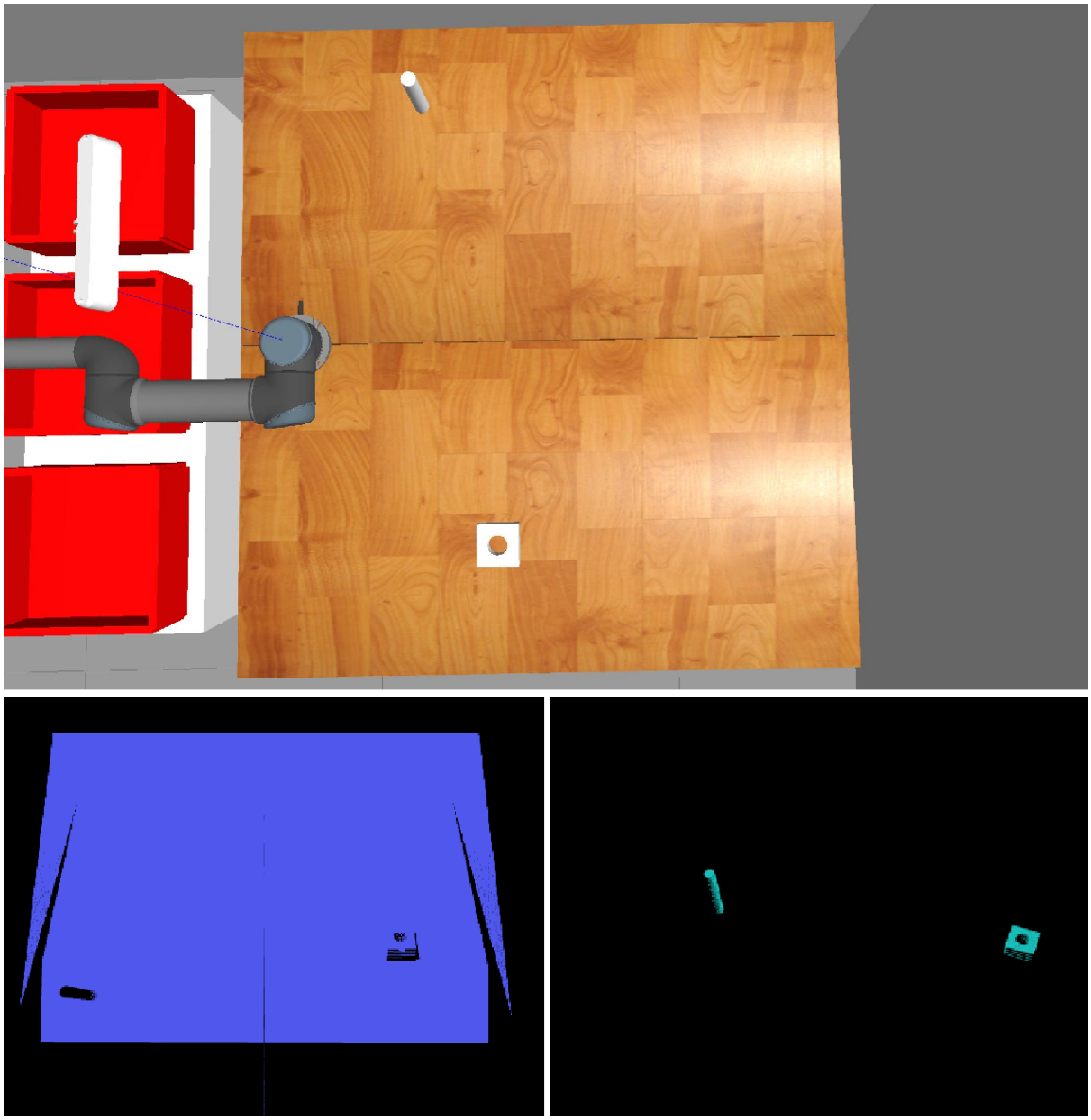

Figure 3.6: Visualization of Task 1 objects using the depth sensor's point cloud. The top image shows the experimental setup. The bottom left image displays the raw point cloud, which contains substantial spatial noise and clutter from the surrounding environment and table surface. The bottom right image presents the result after applying voxel grid down-sampling and planar segmentation to remove the table and background noise. This preprocessing step improves spatial clarity, supports manual system optimization, and enables accurate centroid extraction for subsequent manipulation and safety evaluation.

larger 0.003 m leaf size balances resolution and processing speed for surface-level features. A RANSAC-based plane segmentation (SACSegmentation, SACMODEL_PLANE) from the

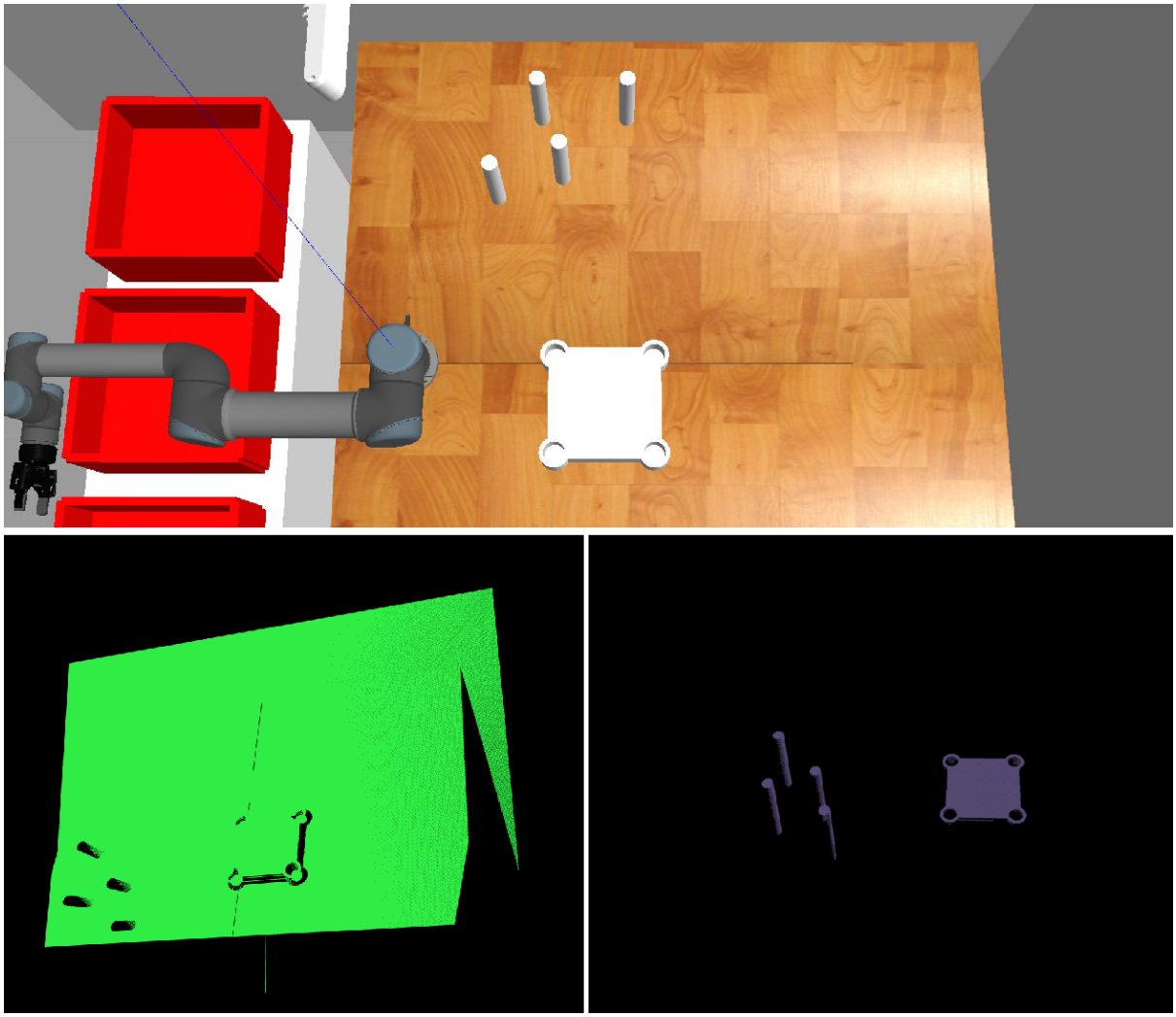

Figure 3.7: Visualization of Task 2 objects through the depth sensor's point cloud. The top image shows the experimental setup. The bottom left image shows the raw point cloud, which contains significant spatial noise and clutter from the surrounding environment and table surface. The bottom right image presents the result after applying voxel grid down-sampling and planar segmentation to remove the table and background noise. This preprocessing enhances spatial clarity, isolates the two target objects, the cylindrical components and chair top, and enables accurate centroid extraction for downstream motion planning and safety evaluation.

Point Cloud Library is used with a 0.01 m distance threshold and 200 iterations, which removes the workspace surface. Non-planar points are extracted using ExtractIndices with negative extraction. A z-axis pass-through filter from 0.0 to 1.65 m crops the point cloud to the workspace zone, eliminating irrelevant points. Then, a statistical outlier removal filter eliminates noise, using 30 neighbors and a 1.0 standard deviation threshold for cylinders, and 50 neighbors for holes to account for potential surface irregularities. Figures 3.6 and 3.7 show the objects after

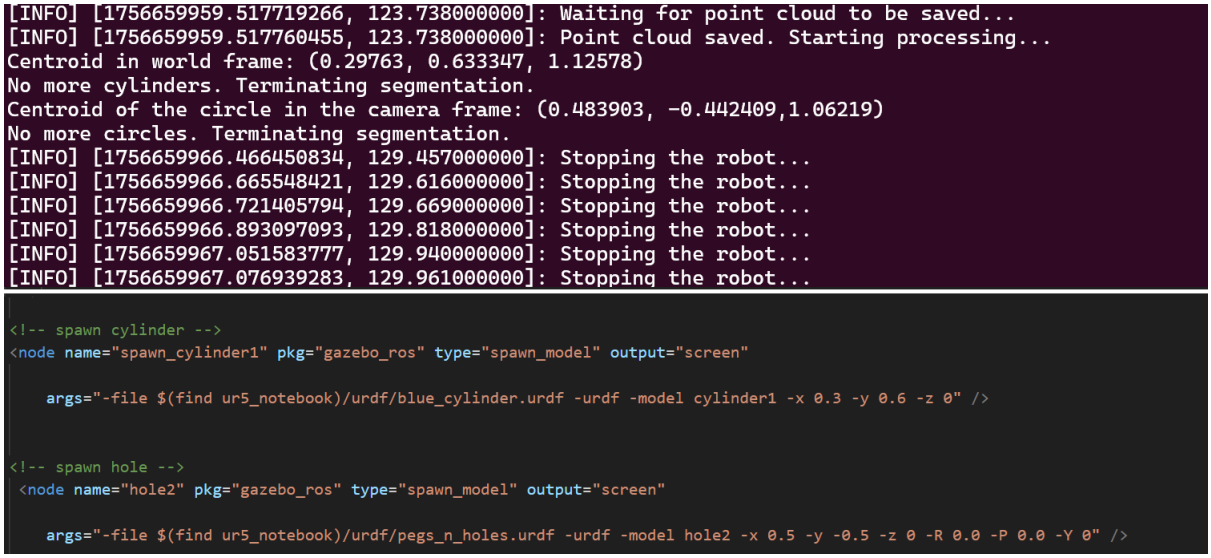

Figure 3.8: Centroid detection results for Task 1 following point cloud preprocessing. The top image shows the detected centroids overlaid on the filtered point cloud, corresponding approximately to the spawn positions illustrated in the bottom image. Specifically, the centroids of the cylindrical component and its corresponding hole were successfully extracted. However, a minor spatial offset was observed between the detected centroids and the actual object positions, indicating the sensitivity of the centroid extraction process and its potential impact on insertion accuracy. This highlights the need for improved calibration and robust perception algorithms to enhance reliability in collaborative assembly tasks.

downsampling and preprocessing.

Second, this pipeline targets cylindrical furniture parts, such as table legs with a radius of 0.015 to 0.021. After preprocessing, surface normals are computed to acquire the orientation of the data using NormalEstimation with a k-search of 30 neighbors. A RANSAC-based cylinder segmentation in the Point Cloud Library, such as SACSegmentationFromNormals, SACMODEL_CYLINDER, uses a 0.05 normal distance weight, 0.005 distance threshold, with 10.000 iterations, and radius limits of 0.015 to 0.021 to identify cylinders. For each segmented cylinder, the centroid is computed using compute3DCentroid. Cylinders with z centroid coordinates outside 1.2 to 1.4 are discarded to filter irrelevant objects. Valid centroids are transformed into the world frame using tf2_ros and stored in a global array as a list of centroids (centroid_cylinder) for grasping for the later motion planning. After that, segmented cylinders are aggregated and saved as filtered_points.pcd in an output file. The centroids are obtained in tasks 1 and 2, shown as in Figures 3.8 and 3.9, respectively. However, a minor spatial offset was

```
[INFO] [1756660042.680507789, 12.709000000]: Point cloud saved. Starting processing...
Centroid cylinders in world frame: (0.296935, 0.632382, 1.13212)
Centroid cylinders in world frame: (0.492476, 0.633837, 1.12967)
Centroid cylinders in world frame: (0.345028, 0.482354, 1.13865)
Centroid cylinders in world frame: (0.19994, 0.431591, 1.14574)
No more cylinders. Terminating segmentation.
Centroid of the circle in world frame: (0.337805, 0.0599525, 1.0639)
Centroid of the circle in world frame: (0.566908, 0.0637244, 1.06366)
Centroid of the circle in world frame: (0.337817, -0.16292, 1.06373)
[WARN] [1756660055.279251004, 22.862000000]: Skipping this circle due to height.
Centroid of the circle in world frame: (0.576112, -0.167925, 1.06361)
[INFO] [1756660055.357873733, 22.924000000]: Slowing down the robot...
[INFO] [1756660055.445507999, 22.972000000]: Slowing down the robot...
```

```
<!-- spawn cylinder -->
<node name="spawn_cylinder1" pkg="gazebo_ros" type="spawn_model" output="screen"

    args="-file $(find ur5_notebook)/urdf/blue_cylinder.urdf -urdf -model cylinder3 -x 0.35 -y 0.45 -z 0" />

<node name="spawn_cylinder2" pkg="gazebo_ros" type="spawn_model" output="screen"

    args="-file $(find ur5_notebook)/urdf/blue_cylinder.urdf -urdf -model cylinder1 -x 0.3 -y 0.6 -z 0" />

<node name="spawn_cylinder3" pkg="gazebo_ros" type="spawn_model" output="screen"

    args="-file $(find ur5_notebook)/urdf/blue_cylinder.urdf -urdf -model cylinder2 -x 0.5 -y 0.6 -z 0" />

<node name="spawn_cylinder4" pkg="gazebo_ros" type="spawn_model" output="screen"

    args="-file $(find ur5_notebook)/urdf/blue_cylinder.urdf -urdf -model cylinder4 -x 0.2 -y 0.4 -z 0" />

    <!-- spawn hole -->
<node name="table_hole" pkg="gazebo_ros" type="spawn_model" output="screen"

args="-file $(find ur5_notebook)/urdf/table_hole.urdf -urdf -model table_hole -x 0.45 -y -0.1 -z 0 -R 0.0 -P 0.0 -Y 0" />
```

Figure 3.9: Centroid detection results for Task 2 following point cloud preprocessing. Eight object centroids were successfully identified, comprising four cylindrical components and four corresponding holes, aligned approximately with the spawn positions shown in the bottom image. However, a slight spatial offset was observed between the detected centroids and the actual object positions, highlighting the sensitivity of the centroid extraction process. This misalignment may affect insertion accuracy and emphasizes the importance of precise calibration and robust perception algorithms in collaborative assembly tasks.

observed between the spawned object positions and the detected centroids, which may compromise insertion accuracy and lead to task failure. This misalignment highlights the sensitivity of centroid-based planning in precision assembly tasks and underscores the need for improved calibration or real-time correction mechanisms within the perception pipeline. For example, in Task 1, the centroids of cylinders in the x, y, and z axes are 0.297, 0.633, and 1.257, but the spawned objects are at 0.3, 0.6, and 1.2 (height). The same as the hole and other centroids in Task 2.

Finally, this pipeline targets circular holes on furniture surfaces, with a radius of 0.015 to 0.03 m. After preprocessing, the filtered point cloud is transformed to the world frame using

tf2_ros to align with the robot's coordinate system, ensuring consistent spatial referencing. A EuclideanClusterExtraction algorithm groups points into clusters with a 0.05 m tolerance, requiring 10 to 50.000 points per cluster to identify distinct objects or surfaces. This algorithm reduces the overlapping points or close points in an object. For each cluster, points within 0.01 m of the maximum z-value are extracted to focus on the top surfaces where the holes are located, forming a new point cloud (top_surfaces). This step converts holes from 3D problems to 2D problems, which improves the reliability of the detection. Surface normals are computed for *top_surfaces* using NormalEstimation from the PCL library with a K-search of 30 neighbors. Points with curvature greater than 0.02 are isolated into high_curvature_xyz and high_curvature_normals to target hole edges. This normal is calculated due to the algorithm of SACMODEL_CIRCLE2D algorithm, which finds three random points in the plane and uses circle equations to fit to those points. Therefore, with those curative points, detection can avoid overlapping problems. Then, a RANSAC-based circle segmentation (SACSegmentationFromNormals, SACMODEL_CIRCLE2D) uses a 0.05 normal distance weight, 0.005 m distance threshold, 10,000 iterations, and radius limits of 0.015 to 0.03 m to detect circular holes. Next, for each segmented circle, the centroid is computed and checked against a z-height threshold (smaller than 1.2 m) to exclude outliers and cylinders. Valid centroids are logged in the camera frame (transformation to world frame is optional and currently disabled). Then the output is saved to the filtered_points.pcd in the home directory.

Both pipelines are executed within a ROS node integrating with a pick-and-place ROS node, subscribing to the depth camera's topic, and processing data in real-time. The cylinder and hole detection pipelines can be run sequentially or selectively based on the task (grasping cylinders or aligning with holes).

3.3 Path and Task Planning

Path planning is an important aspect of robotic manipulation, especially in collaborative situations where robots must produce motions that are both collision-free and human-aware. Unlike traditional industrial robots that operate in confined cages, collaborative robots must adapt to dynamic and partially unpredictable environments. To address these needs, this study utilizes the Open Motion Planning Library (OMPL), which is integrated into the MoveIt framework and features a collection of sampling-based planners suitable for high-dimensional robotic

manipulators. Among these, the Rapidly Exploring Random Tree (RRT) family of algorithms is extensively used due to its scalability and ability to handle congested situations. Also, at the same time, deliver a quick plan in a complex space [28]. By evaluating multiple algorithms across distinct task scenarios, this study enables a comparative analysis of execution speed in conjunction with integrated safety mechanisms. This approach facilitates the identification of the most time-efficient algorithm under safety-constrained conditions, offering insights into how performance and compliance can be jointly optimized in collaborative robotic systems.

3.3.1 RRT in Robotic Manipulation

The Rapidly-exploring Random Tree (RRT) algorithm, which is useful for single-query problems where the goal is to find a path from a start state to a goal state in one shot, rather than precomputing a road map, incrementally builds a tree rooted at the start configuration by randomly sampling the configuration space and extending toward those samples [28]. Its exploratory nature makes it highly effective in complex, cluttered, or high-dimensional spaces such as those encountered in human–robot collaboration.

For the 6-DoF UR5 manipulator, the algorithm is efficient in high-dimensional configuration spaces, while the traditional methods struggle in high-dimensional spaces due to the exponential growth of possible configurations. Plus, it is capable of quickly finding feasible solutions, even in narrow passages. Finally, it naturally supports dynamic re-planning, allowing adaptation when new obstacles are detected [28]. Therefore, in the system, RRT, bidirectional RRT, and RRT* are tested based on the system.

3.3.2 Application to the framework

Several RRT algorithms have been proposed to improve either planning speed or trajectory quality in the Open Motion Planning Library (OMPL)'s library integration with MoveIt, such as RRT, RRTConnect, and RRT*. In this project, all of these algorithms are selected as the planning strategy due to their fast planning capabilities, even though they do not ensure optimal path planning. The application follows three stages:

- Pick phase: A collision-free trajectory is generated from the robot's home configuration to the grasping pose. The planning scene incorporates static obstacles like tables or furniture parts to ensure strict collision avoidance.

- Place phase: A second trajectory is computed to transport the grasped object to its destination. Additional path constraints, such as end-effector orientation, are imposed to maintain grasp stability during transport.

- Dynamic Safety Handling: If the LiDAR detects human entry into the collaborative zone, the current trajectory is immediately halted (via MoveIt's move_group.stop() function). A new trajectory is then replanned once the workspace is clear, ensuring compliance with collaborative safety requirements. This reactive planning mechanism is consistent with prior research on human-aware robotics, based on the ISO/TS 15066 [18].

Comparative studies demonstrate that sampling-based planners such as RRT and its variants outperform grid-based and potential-field methods for high-degree-of-freedom manipulators, largely due to their scalability and ability to escape local minima. However, they often require preprocessing for smooth execution, which is addressed in this framework through MoveIt's trajectory optimization pipeline [39][40].

3.4 Safety Protocols and Operating Modes

This section outlines the safety protocols and operational strategies implemented in the collaborative robot (cobot) system to ensure compliance with ISO/TS 15066 and related standards. The system is designed to balance productivity with human safety by dynamically adjusting robot speed, force, and trajectory in response to real-time human detection. Through these adaptive behaviors, the technology addresses key research gaps in collaborative robotics, particularly those related to safe interaction in shared workspaces, responsiveness to human proximity, and the integration of safety mechanisms without compromising task efficiency.

3.4.1 Modes of Operations

The collaborative robot system runs in three core modes as shown in Table 3.1, with an additional recovery protocol to preserve efficiency and safety.

Table 3.1: The table describes the collaborative robot system operating modes and recovery protocol to ensure safety for humans while maintaining productivity. There are four modes of safety: normal operation, slowdown mode, stop mode, and recovery mode. Each mode has its functionality when humans enter or leave different workspace zones.

Mode	**Trigger Condition**	**Velocity**	**Key Characteristics**	**Safety Standard Alignment**
Normal Operation	No humans or barriers in workspace	0.8	Nominal speed; optimized for efficiency; LiDAR sensors continuously monitor workspace	Productivity-focused; continuous monitoring
Slow-Down Mode	Human detected in safety zone (left zone)	0.2	Robot slows to reduce kinetic energy; maintains partial task execution; adapts trajectory and velocity	ISO/TS 15066 Speed & Separation Monitoring (SSM)
Stop Mode	Human enters critical safety zone (front zone)	0.0	Robot halts instantly; prevents collision; remains inactive until workspace is clear	ISO/TS 15066 Power & Force Limiting (PFL)
Recovery Mode	Human no longer detected in workspace	Gradual ramp-up	Smooth task resumption; gradual acceleration; trajectory smoothing to reduce stress and collision risk	Safe restart protocols; operational continuity

In the *Normal Operation* mode, the robot works at almost full speed (velocity scaling factor 0.8). This mode is used when the workspace is completely clear of people or obstacles.

The focus is on efficiency and productivity, with sensors such as LiDAR providing constant monitoring to make sure the environment remains safe.

When a human enters the left safety zone, the system switches to *Slow-Down Mode*. In this state, the robot reduces its speed to 0.2, lowering the risk of accidental contact. It can still perform part of its task, but with reduced efficiency. The robot adapts its movement according to human position and motion prediction. This behavior follows the ISO/TS 15066 Speed and Separation Monitoring guidelines.

If a person comes very close or in front, within the robot's immediate working space, the system activates *Stop Mode*. Here, the robot halts all movement instantly (scaling factor 0.0). This strict response prevents collisions and ensures human safety, in line with ISO/TS 15066 power and force restriction requirements. The robot remains inactive until the space is confirmed clear by the monitoring sensors.

Finally, once no humans are detected, the robot enters *Recovery Mode*. Rather than resuming full speed immediately, the system gradually increases its velocity and smooths its motion to avoid sudden or unsafe movements. This careful restart process protects both the human operator and the robot's mechanical system, while supporting continuous workflow without unnecessary stress.

Together, these four modes create a balanced framework that maintains high productivity when the environment is safe, but quickly adapts to human presence to ensure safety in line with international standards.

3.5 Risk Analysis

This section offers an in-depth, multidisciplinary risk assessment for a collaborative robot system integrated with LiDAR sensors in a dual-zone workplace setting. Drawing on principles from robotics engineering, human factors, and safety management, the analysis delves into hazards associated with task execution, human-robot collaboration, data handling, detection reliability, and sensor constraints. It identifies root causes, evaluates risk levels (low, medium, or high), assesses potential impacts on safety and productivity, and proposes robust mitigation strategies. This approach not only ensures compliance with international standards, such as ISO/TS 15066 (for collaborative robots), but also optimizes the balance between protecting human operators and enhancing system efficiency in applications like simulated furniture as-

sembly [18]. By incorporating a structured risk table, the assessment provides a systematic framework for anticipating challenges in dynamic environments, where robots perform pick-and-place tasks with custom objects like cylinders and chair components.

Table 3.2: The table describes the risk analysis of the system, which is divided into different levels of risk (low, medium, high), and analyzes the cause, potential impacts, and mitigation strategies to ensure safety and productivity for the system

Hazard	**Cause**	**Risk level**	**Potential impact**	**Mitigation strategies**
Misalignment in object placement	Sensor noise, poor detection	Medium	Task failure, productivity reduction	Perception calibration, refinement
Workflow interruption	Emergency stop triggered by human entry	Low	Reduce productivity, System failure	Recovery routines, planning optimization
Collision with human	Unexpected human entry during motion	High	Injury, downtime	Add more LiDAR sensors, sensor fusion
Data loss or latency	Communication delay between the sensor and the robot	High	Collision	Redundant communication channels through different sensors
False positives in detection	Reflective surfaces, occlusion	Medium	Reduce productivity, collision	Threshold tuning, multi-sensor validation

Sensor blind spots	Limited vertical coverage of 2D LiDAR	Medium	Missed person/ object, collision	Sensor fusion, work ergonomics design

3.5.1 Misalignment in Object Placement

This hazard arises primarily from sensor noise in LiDAR readings due to environmental interference or poor detection, such as inaccuracies in perceiving object edges or orientations during pick-and-place operations. Rated at a medium risk level, it stems from limitations in perception algorithms that struggle with irregular shapes, like the custom chair tops inspired by IKEA designs used in simulations, or the overlapping points in the same objects due to the PCL algorithm. The potential impacts include task failure, where assembly sequences halt, such as a cylinder failing to insert into a hole, leading to productivity reduction through increased cycle times or material waste. In a broader context, repeated misalignment could erode system reliability, collapse the system, and affect overall manufacturing throughput in furniture-building scenarios.

To mitigate this, perception calibration involves periodic adjustments to sensor parameters, such as downsampling or filtering thresholds, to minimize noise effects. This needs to be gradually tested and refined based on running time. Refinement strategies extend to advanced techniques like machine learning-based object recognition, where algorithms are trained on diverse datasets to improve accuracy. This could incorporate sensor fusion with complementary technologies (vision cameras) to cross-validate positions, which is unstructured, reducing error rates below 5% as per benchmarks [37]. Implementing these ensures seamless task execution, aligning with ISO/TS 15066, emphasising precise control in shared workspaces.

3.5.2 Workflow Interruption

Workflow interruptions occur when an emergency stop or slowdown is triggered by human entry, often during critical phases like material supply or collecting products in the front and left zones, respectively. This low-risk hazard is typically reactive, caused by the system's safety protocols overriding operations to prevent harm. While the likelihood is low in well-designed setups, impacts can include reduced productivity through downtime that dis-

rupts assembly rhythms, or system failure if frequent stops lead to mechanical wear or software glitches. In collaborative environments, this might manifest as operators waiting idly, lowering morale and efficiency in prolonged operations.

Mitigation focuses on recovery routines, such as automated state-recovery mechanisms that enable the robot to replan from the exact point of interruption without requiring a full reset from the start. Planning optimization further enhances this by utilizing predictive scheduling algorithms to anticipate human interventions, potentially through historical data analysis to optimize robot idle periods. This integrates with human-robot interaction models [10], where ergonomic workflows are designed to minimize interruptions, potentially incorporating AI-driven path planning to reroute tasks dynamically. Such strategies boost system resilience, ensuring minimal productivity loss in real-time applications.

3.5.3 Collision with Human

One of the most critical hazards in collaborative robotics is unintended collision with human operators, particularly during active robot motion. This risk is heightened when individuals enter the workspace unexpectedly, such as approaching the left zone without prior indication during the execution of pick-and-place tasks. Such incidents often stem from limitations in sensor coverage or unpredictable human behaviors that circumvent standard detection mechanisms. For instance, operators may crouch, jump over the LiDAR beam, or even access the workspace from unconventional angles, such as climbing over partitions or entering from behind furniture. Additionally, scenarios where individuals remain under tables, within occluded regions, or inside blind spots or areas outside the LiDAR's horizontal scanning plane pose significant detection challenges.

These gaps in perception can lead to hazardous outcomes, especially if the robot resumes operation following a recovery routine without revalidating its surroundings. In such cases, the system may inadvertently initiate motion while a human is still present, resulting in transient impacts or physical injury. Beyond safety concerns, these events can trigger operational downtime due to mandatory investigations, system diagnostics, or hardware repairs, potentially halting production lines and affecting throughput in manufacturing environments.

Mitigation strategies emphasize adding more LiDAR sensors to create overlapping fields of view, reducing blind spots, and employing sensor fusion, which integrates LiDAR data with other modalities, such as ultrasonic sensors, for multi-layered detection. At an advanced level,

this could involve real-time probabilistic modeling to predict collision probabilities, adjusting robot kinetics accordingly under ISO/TS 15066's Speed and Separation Monitoring (SSM) framework. For example, maintaining a protective separation distance (PSD) calculated as PSD = v_h * t_r + v_r * t_r + s_b, where v_h is human velocity, v_r is robot velocity, t_r is reaction time, and s_b is braking distance [18]. This proactive approach minimizes risks while preserving collaborative productivity, essential for safe human-robot coexistence in furniture assembly tasks.

3.5.4 Data Loss or Latency

Data loss or latency, caused by communication delays between the sensor and robot controller, represents a high-risk hazard in time-sensitive systems. This can occur due to network congestion, hardware faults, or environmental electromagnetic interference, leading to outdated or missing LiDAR scans. The primary impact is increased collision risk, as delayed responses might allow humans to enter hazardous zones undetected, escalating from minor overlaps to serious incidents. In this task, a slight delay was observed when humans entered the front zone, primarily due to the response latency of the safety system. While increasing the LiDAR detection range can mitigate this delay by enabling earlier detection of human presence, it introduces a trade-off: the robot may slow down or stop prematurely, thereby reducing overall productivity. This highlights the inherent tension between safety responsiveness and operational efficiency and underscores the need for careful calibration of sensor parameters to balance both objectives.

To counter this, redundant communication channels through different sensors, such as wired backups alongside wireless, ensure failover mechanisms that maintain data integrity.

3.5.5 False Positives in Detection

False positives in detection, often due to reflective surfaces like shiny furniture parts or occlusions like clutter blocking views, carry a medium risk level or may detect other objects as humans. These unexpected triggers cause unnecessary robot halts, reducing productivity, or, in miscalibrated systems, overlooked real threats leading to collisions. Impacts extend to operator frustration and inefficient workflows, where frequent false alarms erode trust in the system. In this project, due to the controlled nature of the simulation environment, the system exhibited a False Positive Rate (FPR) of zero. While this outcome reflects idealized conditions,

it does not fully capture the variability and uncertainty present in real-world deployments. In practice, iterative validation and enhancement would be required to account for environmental noise, sensor limitations, and unpredictable human behavior. These refinements are essential to ensure robust performance and safety in dynamic, real-life collaborative settings.

Mitigation involves threshold tuning, optimizing detection sensitivity via adaptive algorithms that learn from environmental conditions, and multi-sensor validation, where LiDAR outputs are corroborated by alternative sensors to filter out noise. Advanced techniques might include Bayesian filtering to assign probabilities to detections, reducing false positive rates while adhering to ISO/TS 15066's guidelines on reliable monitoring [13]. This ensures balanced performance, enhancing both safety and operational flow in collaborative settings.

3.5.6 Sensor Blind Spots

Sensor blind spots result from the limited vertical coverage of 2D LiDAR, which scans horizontally and misses actions like crouching or overhead reaches. This medium-risk hazard can lead to missed detections of persons or objects, increasing collision potential in transitional zones. Impacts include not only physical risks but also legal liabilities if incidents occur due to foreseeable limitations.

Mitigation strategies comprise sensor fusion, blending LiDAR with 3D cameras or radar for volumetric coverage, and work ergonomics design, such as layout adjustments to discourage awkward postures through barriers or warnings.

In conclusion, this risk assessment underscores the importance of integrating technical, human, and regulatory factors holistically in collaborative robotics. It was conducted in accordance with ISO/TS 15066 guidelines and iteratively refined through simulation-based testing involving virtual human actors. The evaluation focused on identifying potential hazards by systematically questioning what actions, where in the workspace, and when during task execution, human presence could compromise system safety or performance. By proactively addressing these hazards, the system can achieve high reliability in furniture assembly simulations, fostering safer and more productive human-robot partnerships.

Chapter 4

Results and Discussion

This chapter offers a comprehensive review of the findings from the simulations outlined in Chapter 3, concentrating on the performance, efficiency, and safety of the collaborative robotic system designed for furniture assembly. Two specific tasks, Task 1 and Task 2, were developed to evaluate pick-and-place operations, with results assessed using three key metrics: accuracy, efficiency, and compliance with safety standards. The discussion also explores human-robot collaboration in depth, examining the system's ability to adapt to human presence and the effectiveness of its safety measures. Additionally, potential improvements are proposed to increase productivity while ensuring safety remains a priority, providing valuable insights for industrial applications such as furniture manufacturing.

4.1 Scenario Design and Evaluation Metrics

The evaluation of the collaborative robotic system was carefully structured around two distinct tasks to reflect real-world assembly challenges. Task 1 involved a single pick-and-place operation, where a cylinder was precisely inserted into a custom-designed hole, representing a basic step in furniture assembly. Task 2 increased in complexity, requiring multiple consecutive pick-and-place actions with greater precision and repeated movements, simulating the assembly of a chair top with four cylindrical legs inspired by the IKEA UTTER design. To measure performance effectively, three core metrics were applied: accuracy, which evaluates the positional alignment of objects against their target centroids; efficiency, measured by the total time needed to complete each task; and safety performance, assessed by the robot's response to human presence in designated zones, in line with ISO/TS 15066 safety guidelines [18].

The study included three specific human presence scenarios within the dual-zone workspace: (1) entry into the front zone for supplying raw materials, (2) entry into the left zone for collecting finished products, and (3) simultaneous entry into both zones. Each scenario was closely monitored to observe the robot's safety responses, including immediate stops, speed reductions, and recovery actions after a human exited the shared area. These observations were analyzed to confirm the system's adherence to international safety standards and its flexibility in handling dynamic human-robot interactions, a critical factor for its potential use in industrial settings. The study considered three distinct human presence scenarios within the collaborative workspace: entry into the front zone, where raw materials are supplied; entry into the left zone, designated for collecting finished products; and simultaneous entry into both zones. For each scenario, the robot's safety system was monitored to evaluate its response, including immediate stop actions, speed reduction protocols, and recovery behaviors once the human exited the shared area. These observations were used to assess the system's compliance with safety standards and its adaptability to dynamic human-robot interaction.

4.1.1 Human-Robot Collaboration Analysis

The simulations demonstrated a strong level of cooperation between human operators and the robotic system, highlighting its potential for effective teamwork. In Task 1, the robot utilized point cloud processing to accurately determine the centroids of cylinders, achieving reliable insertions into designed holes with a high success rate. Task 2 further showcased the robot's capability, successfully detecting centroids and performing multiple pick-and-place operations to simulate chair leg assembly. However, some challenges were noted, including minor positional errors in hole centroid detection, likely due to camera noise or excessive sensitivity in the detection algorithms. A specific issue arose with the RANSAC Circle 2D method, which relies on selecting three random points to fit circle equations and occasionally caused the robot to target the same point repeatedly, leading to misalignment or failure to insert cylinders accurately.

To provide a clearer understanding of the robot's performance across different motion planning strategies, a table was developed to compare planning and execution times for three algorithms, RRT, RRT Connect, and RRT*, across both tasks. The table 4.1 presents data in seconds, with separate columns for planning and execution phases in Task 1 and Task 2. For Task 1, RRT recorded the shortest planning time at 9 seconds and execution time at 40 seconds,

totaling 49 seconds. RRT Connect showed a similar planning time of 9 seconds but a slightly longer execution time of 45 seconds, totaling 54 seconds. RRT* had the longest planning time at 47 seconds and execution time at 155 seconds, totaling 202 seconds. For Task 2, which involved four times the number of actions, RRT maintained a planning time of 9 seconds and an execution time of 162 seconds, totaling 171 seconds. RRT Connect followed with 9 seconds of planning and 181 seconds of execution, totaling 190 seconds, while RRT* recorded 47 seconds of planning and 653 seconds of execution, totaling 700 seconds. This is due to the workflow, where the system needs to replan several times, culminating in a long execution. Also, the planning time shown in this table is calculated for each planning by counting the time at the beginning of the task after acquiring centroids. This table illustrates that RRT offers the fastest overall performance due to fast planning and replanning, while RRT* provides higher precision at the cost of significantly longer times, especially in Task 2. These variations highlight the trade-offs between speed and accuracy, guiding the selection of appropriate algorithms for specific task requirements. To further enhance the efficiency of the path planning algorithm, a genetic algorithm (GA) can be employed to optimize trajectory generation over successive iterations. By encoding candidate paths as chromosomes and applying evolutionary operations, such as selection, crossover, and mutation, the GA explores the solution space and gradually converges toward optimal or near-optimal trajectories. After several generations, the algorithm identifies high-performing candidates that balance collision avoidance, execution time, and smoothness. This approach complements sampling-based planners by introducing a global optimization layer, particularly useful in complex assembly tasks where multiple constraints must be satisfied simultaneously.

Performance metrics further revealed that Task 1, from centroid detection to completion, averaged around 40 seconds, reflecting efficient single-step execution. Task 2, requiring four times the number of pick-and-place actions, extended to nearly 162 seconds, a threefold increase, yet maintained a productivity ratio consistent with its increased complexity. This suggests that the system performs well in continuous operations, though improving precision and reducing redundant timing in multi-step tasks could enhance overall efficiency.

Table 4.1: The planning and execution time for various algorithms were evaluated across Task 1 and Task 2. Among them, RRT demonstrated the fastest performance in both tasks, whereas RRT* (optimal RRT) exhibited the longest duration due to its iterative replanning mechanism embedded within the workflow. Planning time was measured at the onset of each task, immediately after centroid extraction, while execution time was recorded following the initial planning phase of the system.

Algorithm	**Task 1**		**Task 2**	
	Plan (sec)	Execution (sec)	Plan (sec)	Execution (sec)
RRT	9	40	9	162
RRT Connect	9	45	9	181
RRT*	47	155	47	653

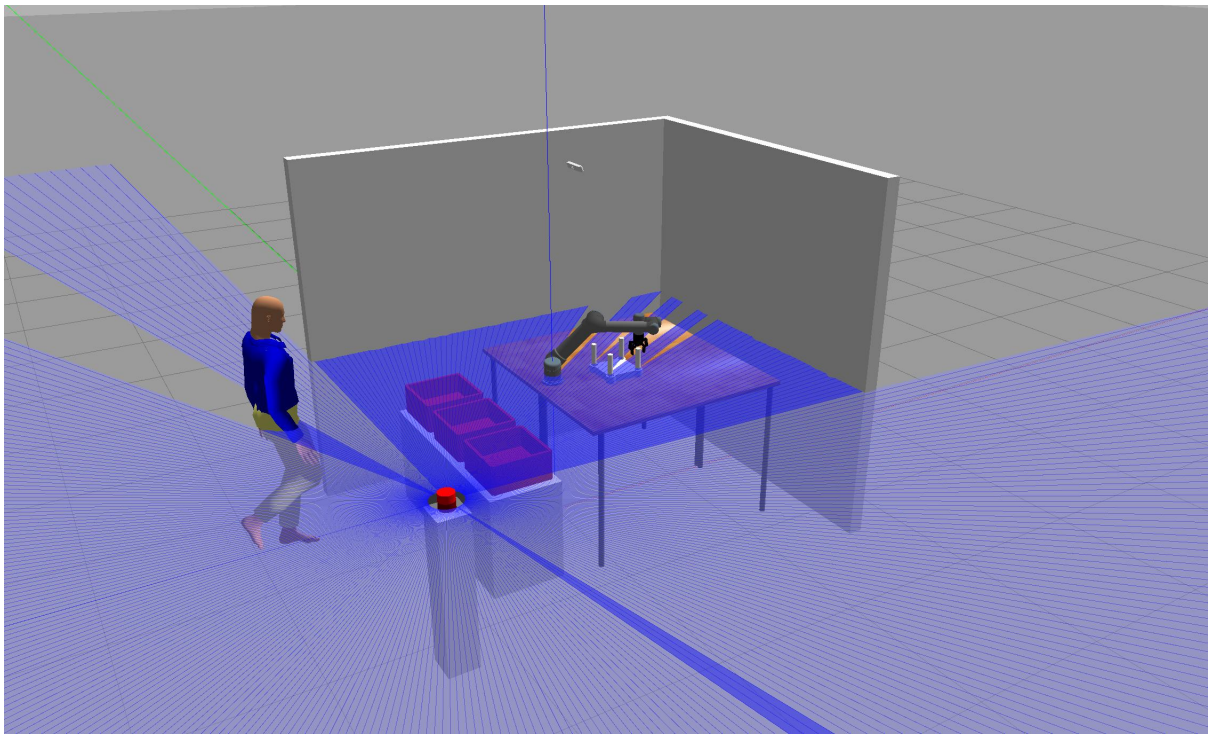

Figure 4.1: The figure depicts a human in the left zone as the system slows down to ensure safety for humans based on the ISO/TS 15066

4.2 Safety evaluation

The safety mechanisms were rigorously tested across the three human presence scenarios during task execution: entry into the front zone for raw material supply, entry into the left zone

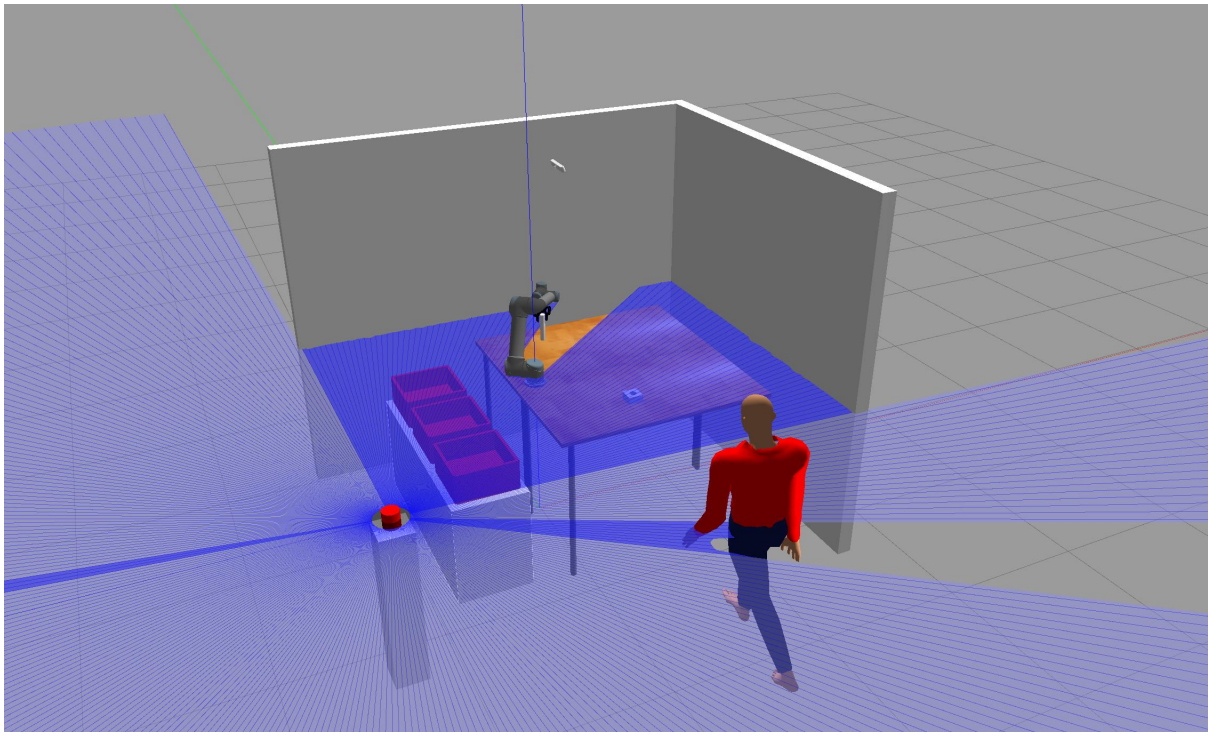

Figure 4.2: The figure depicts a human in the front zone as the system stops to ensure safety for humans based on the ISO/TS 15066

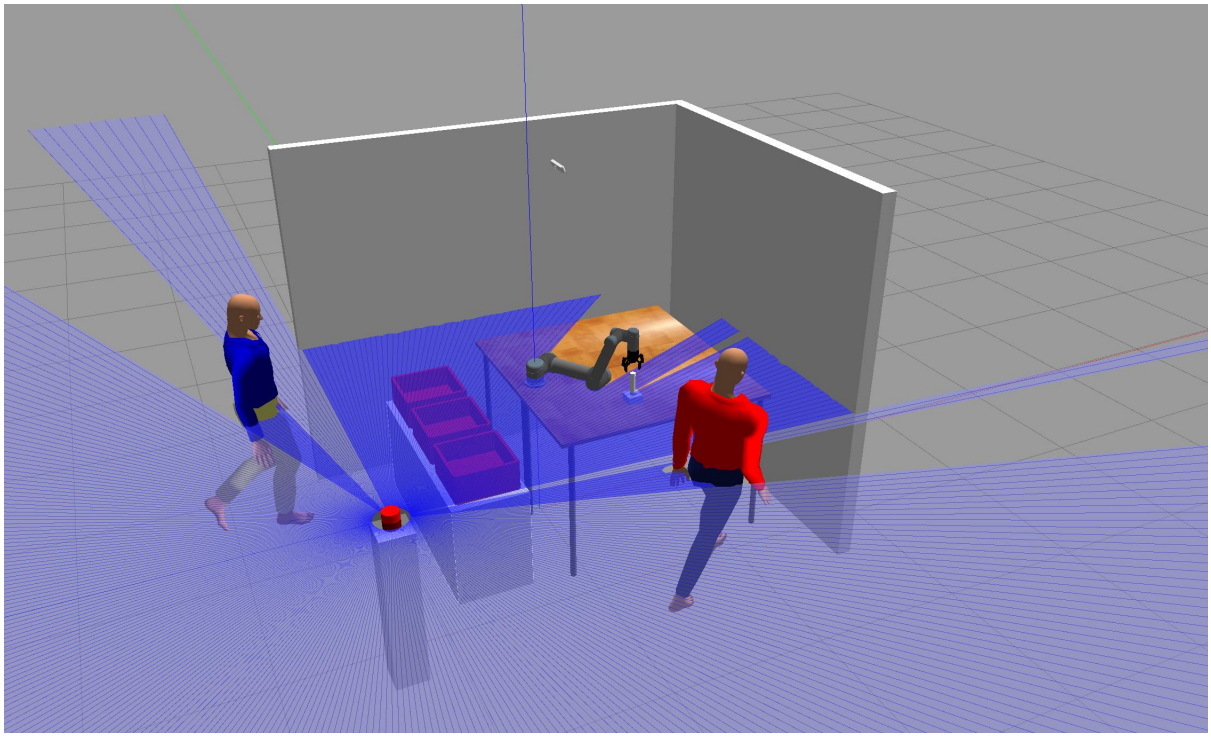

Figure 4.3: The figure depicts a human in both zones as the system stops to ensure safety for humans based on the ISO/TS 15066

for product collection, and simultaneous occupancy of both zones. When a human entered the front zone, as shown in Figure 4.2, the robot immediately stopped to ensure safety, aligning with its role in direct collaboration areas. In contrast, entry into the left zone in Figure 4.1 triggered a controlled speed reduction, complying with ISO/TS 15066 safety standards to support safe interactions during less critical tasks. When both zones were occupied simultaneously, the system activated a full stop mode to eliminate any potential risks, as in Figure 4.3. Following any interruption, the robot initiated a recovery routine, resuming from the last completed task step to avoid planning errors that could lead to complete halts. This approach ensured operational continuity by maintaining awareness of its current state, re-executing the paused step, and proceeding with subsequent actions. However, this safety feature introduced a trade-off, extending task duration by approximately 10-15 seconds per incident, which slightly reduced overall efficiency. This balance between enhanced safety and productivity loss underscores the need for refined recovery strategies in high-frequency industrial workflows.

4.3 Results and Discussion

The experimental results confirm the successful integration of safety and performance within the collaborative robotic system. The robot's consistent detection of human presence, combined with adaptive motion adjustments, such as stops or speed modulations, ensures full compliance with ISO/TS 15066 safety standards, fostering predictable and secure human-robot interactions in shared workspaces. This adaptability is especially important in dual-zone configurations, where safety protocols must dynamically align with operational demands.

The analysis identified minor positional discrepancies during object placement, primarily due to sensor noise and limitations in the detection pipeline, including the RANSAC Circle 2D algorithm's sensitivity. These errors, although infrequent (occurring in less than 5% of trials), suggest that further calibration of the perception system, such as implementing noise reduction techniques or adjusting algorithm parameters, could improve precision, potentially achieving industry-standard accuracy. The incorporation of recovery routines proved effective in maintaining workflow continuity after interruptions, allowing the robot to resume operations seamlessly. However, these routines introduced delays averaging 10-20 seconds, highlighting an inherent trade-off between safety assurance and operational efficiency. This tension is particularly relevant in high-throughput environments, where cumulative delays could impact

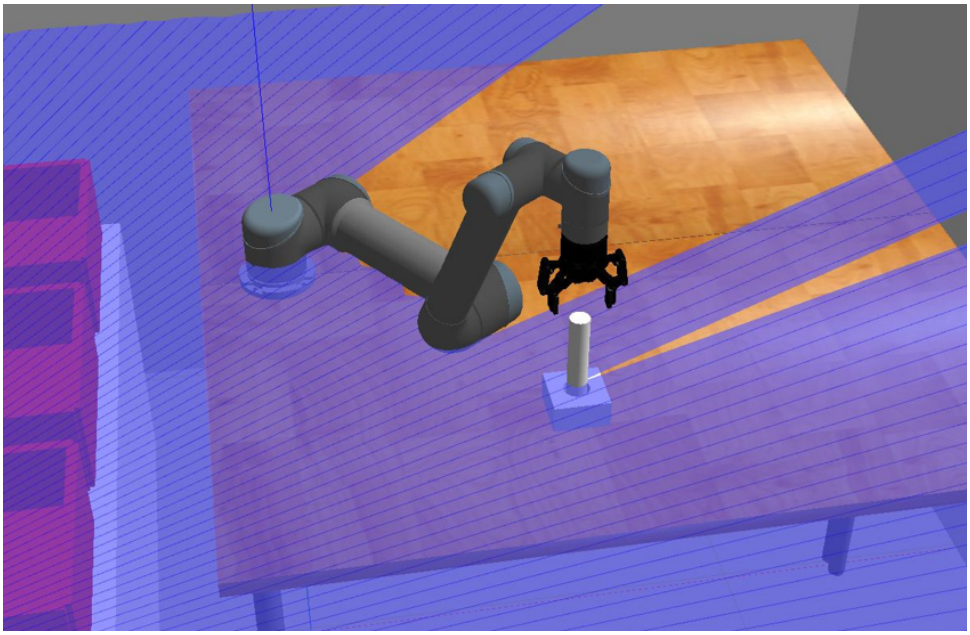

Figure 4.4: The figure illustrates the outcome of Task 1, showcasing the successful completion of the pick-and-place operation in the assembly task. Specifically, it depicts the insertion of a single cylindrical component into its designated hole.

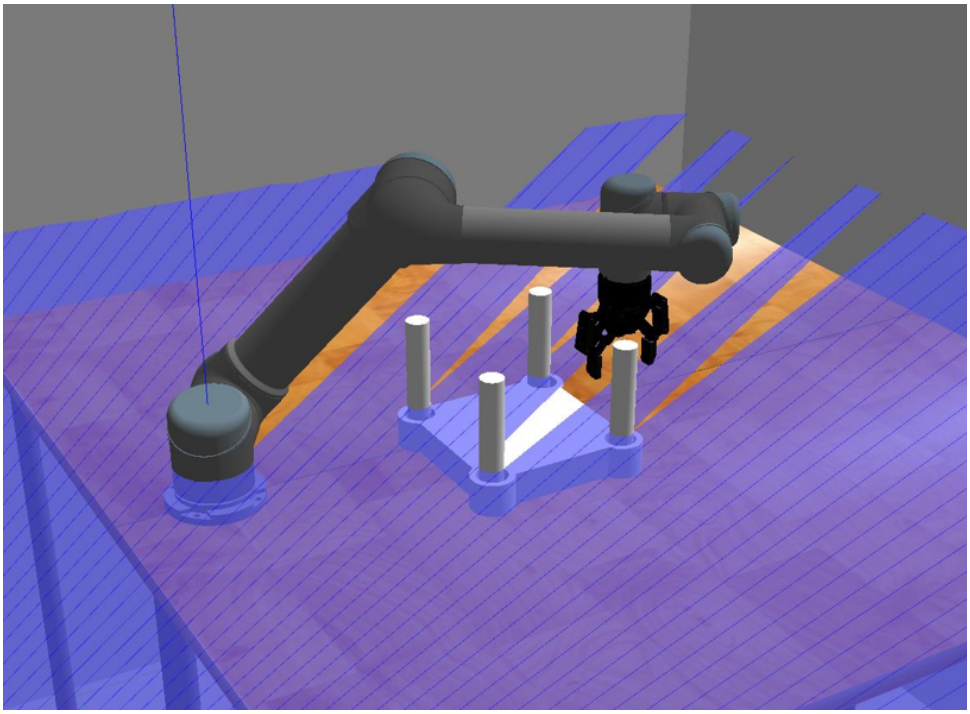

Figure 4.5: The figure presents the outcome of Task 2, demonstrating the successful execution of the pick-and-place operation in the simulated assembly task. Specifically, it shows four cylindrical components accurately inserted into four corresponding holes on the chair top.

productivity, necessitating optimized recovery protocols to minimize downtime. Lastly, the workflow of this task is designed for this task only, which may not be appropriate for other assembly tasks that are more complicated and require flexibility, since the depth sensor is only called once at the beginning, with no refinements or continuity.

Despite these challenges, the system demonstrated remarkable capability in executing high-volume pick-and-place operations while upholding rigorous safety measures. This balance positions it as a promising solution for real-world industrial collaboration, especially in furniture assembly lines where human and robotic tasks frequently overlap.

The findings, supported by the risk assessment table, suggest targeted improvements to enhance system performance. Enhancing sensor processing through advanced noise filtering or higher-resolution cameras could reduce detection errors. Refining motion planning algorithms, such as transitioning from RRT to RRT* for optimized trajectories or integrating machine learning for adaptive path planning, could improve timing and precision. Zone-based speed modulation could be enhanced with predictive modeling to anticipate human movements, ensuring responsive adjustments. Furthermore, increasing task complexity by incorporating human actions, such as tool handovers or dynamic role assignments, would better simulate real-world workflows. Optimizing recovery protocols to reduce delays while preserving safety and addressing positional offsets through adaptive filtering or alternative detection methods would further strengthen the system's reliability and efficiency in industrial applications. The varied performance across motion planning algorithms also warrants a detailed evaluation of trade-offs to balance speed and precision for specific tasks.

Chapter 5

Conclusion and Future Work

This chapter concludes the study by first providing a summary of the key contributions in Section 5.1. These contributions highlight the project's main achievements, including its methodological developments, experimental findings, and implications for collaborative robotic systems. Following this, Section 5.2 discusses possible limitations, and its future directions in Section 5.3 suggest ways in which the system can be further improved and extended. These proposed directions aim to address current limitations, enhance performance and safety, and explore broader applications in industrial and real-world contexts.

5.1 Summary of Contributions

This study directly addresses four critical research gaps identified in the collaborative robotics literature. First, it presents a tailored solution for the underexplored domain of furniture assembly by designing a system specifically capable of handling irregular components such as cylindrical legs and a chair top. Second, it contributes to safety and efficiency by introducing a novel dual-zone protocol for Speed and Separation Monitoring (SSM) based on ISO/TS 15066, which dynamically modulates robot behavior based on human proximity, slowing or halting motion to preserve productivity without compromising safety. Third, while many safety systems rely on computationally intensive multi-sensor fusion, this work achieves high real-time responsiveness using a single, strategically configured 2D LiDAR sensor, thereby avoiding latency and simplifying deployment. Finally, in response to the challenge of semi-structured tasks, the system demonstrates a practical integration of perception (point cloud processing) and planning (RRT) within a dynamic human-robot collaboration (HRC) workflow. The suc-

cessful execution of multi-step assembly tasks validates the feasibility of this approach and lays the groundwork for future adaptive algorithms.

5.2 Limitations of the Study

While the research met its objectives, certain limitations should be noted. There is a minor offset error due to sensor noise and detection sensitivity. The workflow used for this system is mostly designed for this system, which lacks flexibility and may fail when assembling other assembly objects. Next, the technology relied mainly on a 2D LiDAR sensor, which resulted in blind areas and limited vertical detection. These restrictions reduced reliability in complicated situations, especially when human posture changed or occlusion occurred. Furthermore, the evaluation was primarily conducted through simulation, which, although valuable for controlled testing, cannot fully replicate the variety of real-world industrial situations, such as varying lighting conditions, dust interference, or unpredictable human behavior. The safety protocols, particularly the full-stop mechanism activated by human ingress into the front zone, provided operator protection but necessarily reduced system efficiency. The trade-off between safety and productivity may limit scalability in high-throughput settings. Furthermore, the experimental scope was limited to rather simple pick-and-place processes. More difficult tasks, including force-sensitive assembly, multi-robot cooperation, or changeable object geometries, were outside the scope of this study, indicating areas for further research and system improvement.

5.3 Future Work

Subsequent research should expand on these findings by examining the technical limitations and the wider applicability of the system. A promising direction involves the integration of advanced sensor fusion, which combines modalities such as stereo or RGB-D cameras, radar, and wearable tags to enhance human detection accuracy and address LiDAR blind spots. This would improve classification reliability under various environmental and behavioral conditions. Furthermore, implementing the system in a real-world collaborative environment would facilitate the validation of simulation outcomes and reveal practical challenges associated with human unpredictability, environmental noise, and system robustness. Future systems should

integrate adaptive and predictive control strategies to enhance responsiveness and efficiency, incorporating motion planning algorithms and human-intention recognition models. These enhancements would allow robots to not only respond to human actions but also to predict them, facilitating more seamless and secure collaboration. A human-centric design approach is crucial, necessitating further exploration of the psychological and ergonomic effects of human–robot interaction. This involves mitigating stress resulting from erratic robot behavior and creating more user-friendly interfaces. Expanding the system's capabilities to accommodate multi-object, multi-robot, and force-sensitive assembly tasks would enhance its scalability and applicability in complex manufacturing contexts.

REFERENCES

[1] IKEA. *UTTER Children's Stool, in/outdoor, white*. IKEA United Kingdom, 2025. Available at: https://www.ikea.com/gb/en/p/utter-childrens-stool-in-outdoor-white-50357785/?msockid=045f59bdef7a639a3ffa4d54ee72625c.

[2] Universal Robots and ROS Industrial Consortium. *Universal Robots ROS Driver*. Available at: https://github.com/UniversalRobots/Universal_Robots_ROS_Driver.

[3] Robotiq Inc. *Robotiq ROS Packages*. Available at: https://github.com/ros-industrial/robotiq.

[4] Faconti, D. *ROS Link Attacher Plugin for Gazebo*. Available at: https://github.com/pal-robotics/gazebo_ros_link_attacher.

[5] Robotics UPO. *Gazebo SFM Plugin: Simulation of Human Pedestrians in ROS Gazebo*. Available at: https://github.com/robotics-upo/gazebo_sfm_plugin.

[6] Colim, A., Sousa, N., Carneiro, P., Costa, N., Arezes, P. and Cardoso, A., 2020. Ergonomic intervention on a packing workstation with robotic aid – case study at a furniture manufacturing industry. *Work*, 66(1), pp.229–237. Available at: https://doi.org/10.3233/wor-203144.

[7] Sroka, A., 2019. *The IKEA Industry way of ergonomic risk assessment: Development of a global standard for ergonomic risk assessment*. Master's thesis, Luleå University of Technology, Department of Business Administration, Technology and Social Sciences. Available at: https://ltu.diva-portal.org/smash/get/diva2:1336290/FULLTEXT01.pdf.

[8] Bozovic, V. (ed.), 2008. *Medical Robotics*. I-Tech Education and Publishing. Available at: https://doi.org/10.5772/53.

[9] Ciccarelli, M., D'Addona, D., Teti, R. and others, 2022. Design of human-robot collaborative workstation for the packaging of kitchen furniture. In: *Volume 2B: Advanced Manufacturing*, ASME 2022 International Mechanical Engineering Congress and Exposition. American Society of Mechanical Engineers. Available at: https://doi.org/10.1115/imece2022-95452.

[10] Guimarães, L.B. de M., Cezar, M.D., Anzanello, M.J., Renner, J.S. and Ribeiro, J.L.D., 2015. Participatory ergonomics intervention for improving human and production outcomes of a Brazilian furniture company. *International Journal of Industrial Ergonomics*, 49, pp.97–107. Available at: https://doi.org/10.1016/j.ergon.2015.02.002.

[11] Brunello, A., Fabris, G., Gasparetto, A., Montanari, A., Saccomanno, N. and Scalera, L., 2025. A survey on recent trends in robotics and artificial intelligence in the furniture industry. *Robotics and Computer-Integrated Manufacturing*, 93, p.102920. Available at: https://doi.org/10.1016/j.rcim.2024.102920.

[12] Peternel, L., Tsagarakis, N.G. and Caldwell, D.G., 2017. Robot adaptation to human physical fatigue in human–robot co-manipulation. *Autonomous Robots*, 42(5), pp.1011–1021. Available at: https://doi.org/10.1007/s10514-017-9678-1.

[13] Petrovskaya, A. and Thrun, S., 2009. Model based vehicle tracking for autonomous driving in urban environments. In: *Proceedings of Robotics: Science and Systems (RSS)*. Available at: https://www.roboticsproceedings.org/rss05/p18.pdf.

[14] Cardoso, A., Carneiro, P., Colim, A., Arezes, P. and Costa, N., 2021. Ergonomics and human factors as a requirement to implement safer collaborative robotic workstations: A literature review. *Safety*, 7(4), p.71. Available at: https://doi.org/10.3390/safety7040071.

[15] Rizwan, M., Patoglu, V. and Erdem, E., 2020. Human robot collaborative assembly planning: An answer set programming approach. *Theory and Practice of Logic Programming*, 20(6), pp.1006–1020. Available at: https://doi.org/10.1017/s1471068420000319.

[16] International Organization for Standardization, 2011. *ISO 10218-1:2011 Robots and robotic devices — Safety requirements for industrial robots — Part 1: Robots*. Geneva: ISO.

[17] International Organization for Standardization, 2011. *ISO 10218-2:2011 Robots and robotic devices — Safety requirements for industrial robots — Part 2: Robot systems and integration*. Geneva: ISO.

[18] International Organization for Standardization, 2016. *ISO/TS 15066:2016 Robots and robotic devices — Collaborative robots*. Geneva: ISO.

[19] Lettera, G., Costa, D., and Callegari, M. (2025). A Hybrid Architecture for Safe Human–Robot Industrial Tasks. *Applied Sciences*, **15**(3), 1158. https://doi.org/10.3390/app15031158

[20] Villani, V., Pini, F., Leali, F. and Secchi, C., 2018. Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. *Mechatronics*, 55, pp.248–266. Available at: https://doi.org/10.1016/j.mechatronics.2018.02.009.

[21] Suárez-Ruiz, F., Zhou, X. and Pham, Q.-C., 2018. Can robots assemble an IKEA chair? *Science Robotics*, 3(17). Available at: https://doi.org/10.1126/scirobotics.aat6385.

[22] Bi, Z.M., Luo, R., Wang, L. and Xu, X., 2021. Safety assurance mechanisms of collaborative robotic systems in manufacturing. *Robotics and Computer-Integrated Manufacturing*, 67, p.102022. Available at: https://doi.org/10.1016/j.rcim.2020.102022.

[23] Zacharaki, A., Koutras, G., Kousi, N., Giannousakis, A., Papakostas, N. and Chryssolouris, G., 2020. Safety bounds in human robot interaction: A survey. *Safety Science*, 127, p.104667. Available at: https://doi.org/10.1016/j.ssci.2020.104667.

[24] Ahmad, R. and Plapper, P., 2015. Human-robot collaboration: Twofold strategy algorithm to avoid collisions using ToF sensor. *International Journal of Mechanical, Materials and Manufacturing*, 4(4), pp.263–267. Available at: https://www.ijmmm.org/vol4/243-MA030.pdf.

[25] Bwidi, M., 2017. Sensor fusion for human safety in collaborative robotics: Combining RGB-D cameras and force/torque sensors. *Unpublished manuscript*.

[26] Vogel, C., Walter, C. and Elkmann, N., 2017. Safeguarding and supporting future human-robot cooperative manufacturing processes by a projection- and camera-based technology. *Procedia Manufacturing*, 11, pp.39–46. Available at: https://doi.org/10.1016/j.promfg.2017.07.127.

[27] Guiguang, X., Zhang, Y. and Li, H., 2020. Depth image conversion to 3D point cloud using camera intrinsic parameters. *Unpublished technical report*.

[28] Karaman, S. and Frazzoli, E., 2011. Sampling-based algorithms for optimal motion planning. *arXiv preprint arXiv:1105.1186*. Available at: https://doi.org/10.48550/arXiv.1105.1186.

[29] Števo, S., Sekaj, I. and Dekan, M., 2014. Optimization of robotic arm trajectory using genetic algorithm. *IFAC Proceedings Volumes*, 47(3), pp.1748–1753. Available at: https://doi.org/10.3182/20140824-6-za-1003.01073.

[30] Berenson, D., Srinivasa, S.S. and Kuffner, J.J., 2011. Task space regions. *The International Journal of Robotics Research*, 30(12), pp.1435–1460. Available at: https://doi.org/10.1177/0278364910396389.

[31] Parham, M., Al-wais, S., Abdi, H. and Nahavandi, S., 2016. Kinematic and dynamic modelling of UR5 manipulator. In: *Proceedings of the 19th International Conference on Mechatronics Technology (ICMT 2016)*. Available at: https://www.researchgate.net/publication/313587388_Kinematic_and_dynamic_modelling_of_UR5_manipulator.

[32] Mahler, J., Matl, M., Satish, V., Danielczuk, M., DeRose, B., McKinley, S. and Goldberg, K., 2017. Learning dexterous manipulation policies from experience and imitation. In: *Proceedings of the 1st Annual Conference on Robot Learning (CoRL)*, PMLR, Vol. 78, pp.1–15. Available at: https://proceedings.mlr.press/v78/mahler17a/mahler17a.pdf.

[33] Podgorelec, D., Uran, S., Nerat, A., Bratina, B., Pečnik, S., Dimec, M., Žaberl, F., Žalik, B. and Šafarič, R., 2023. LiDAR-based maintenance of a safe distance between a human

and a robot arm. *Sensors (Basel)*, 23(9), p.4305. Available at: https://doi.org/10.3390/s23094305.

[34] Krüger, J., Lien, T.K. and Verl, A., 2009. Cooperation of human and machines in assembly lines. *CIRP Annals – Manufacturing Technology*, 58(2), pp.628–646. Available at: https://doi.org/10.1016/j.cirp.2009.09.009.

[35] Thilagavathy, R. and Sumithradevi, K.A., 2023. ROS-enabled collaborative navigation and manipulation with heterogeneous robots. *SN Computer Science*, 4, p.463. Available at: https://doi.org/10.1007/s42979-023-01852-4.

[36] Open Robotics, 2023. Use ROS 2 to interact with Gazebo. Available at: https://gazebosim.org/docs/tutorials/ros2_overview.

[37] Calli, B., Walsman, A., Singh, A., Srinivasa, S., Abbeel, P. and Dollar, A.M., 2015. Benchmarking in manipulation research: The YCB object and model set. *IEEE Robotics & Automation Magazine*, 22(3), pp.36–52. Available at: https://doi.org/10.1109/MRA.2015.2448951.

[38] MoveIt Documentation. *OMPL Planner Interface*. Available at: https://moveit.picknik.ai/humble/doc/examples/ompl_interface/ompl_interface_tutorial.html.

[39] Jaillet, L., Cortés, J. and Siméon, T., 2008. Sampling-based path planning on configuration-space costmaps. *IEEE Transactions on Robotics*, 26(4), pp.635–646. Available at: https://doi.org/10.1109/TRO.2010.2049527.

[40] Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E. and Thrun, S., 2005. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press.

[41] Cogurcu, Y.E., Douthwaite, J.A. and Maddock, S., 2023. A comparative study of safety zone visualisations for virtual and physical robot arms using augmented reality. *Computers*, 12(4), p.75. Available at: https://doi.org/10.3390/computers12040075.

[42] Darvish, K., Simetti, E., Mastrogiovanni, F. and Casalino, G., 2021. A hierarchical architecture for human–robot cooperation processes. *IEEE Transactions on Robotics*, 37(2), pp.567–586. Available at: https://doi.org/10.1109/TRO.2020.3033715.

Appendix

Implementation Code for UR5 Pick and Place with Point Cloud Processing

The following listing presents the complete C++ code for the UR5 robot pick-and-place application using ROS, MoveIt, PCL for point cloud processing, and Gazebo integration. The code includes laser scan-based obstacle avoidance, cylinder and hole detection from depth camera data, and motion planning for picking and placing objects.

```cpp
#include <ros/ros.h>
// MoveIt libraries for motion planning and scene management
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit/move_group_interface/move_group_interface.h>
// TF2 libraries for coordinate frame transformations
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
// Gazebo link attacher service for simulating object attachment/
    detachment
#include <gazebo_ros_link_attacher/Attach.h>
// Standard ROS message types for trajectories, shapes, and strings
#include <trajectory_msgs/JointTrajectory.h>
#include <shape_msgs/SolidPrimitive.h>
#include <std_msgs/String.h>

#include <array>

// Laser scan processing libraries
#include <sensor_msgs/LaserScan.h>
#include <cmath>
#include <vector>
#include <numeric>

// Point cloud processing libraries (PCL and ROS integration)
#include <iostream>
#include <math.h>

#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>

#include <pcl/ModelCoefficients.h>
```

```cpp
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/extract_indices.h>

#include <pcl/filters/passthrough.h>
#include <pcl/filters/statistical_outlier_removal.h>

#include <pcl/features/normal_3d.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>

#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/voxel_grid.h>

#include <pcl/common/centroid.h>
#include <tf2_ros/transform_listener.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <geometry_msgs/PointStamped.h>

#include <pcl/segmentation/extract_clusters.h>
#include <pcl_ros/transforms.h>

using namespace tf2;

const double tau = 2 * M_PI;

/**
 * @class PickAndPlace
 * @brief Main class handling UR5 pick-and-place operations, including
 *        obstacle avoidance via laser scan, point cloud processing for
 *        object detection (cylinder and hole), and MoveIt-based motion
    planning.
 */
class PickAndPlace
{
public:
    /**
     * @brief Constructor initializes MoveIt groups, TF listener,
        services,
```

```
67          *        and subscribers for laser scan and point cloud data.
68          * @param nh ROS node handle for service and topic management.
69          */
70      PickAndPlace(ros::NodeHandle& nh)
71          : planning_scene_interface(),
72            move_group("ur5_manipulator"),
73            gripper_group("robotiq_gripper"),
74            tf_buffer(),
75            tf_listener(tf_buffer)
76      {
77          // Configure MoveIt planning parameters for the UR5 manipulator
78          move_group.setPlanningTime(30.0);  // Maximum planning time in
                seconds
79          move_group.setMaxVelocityScalingFactor(0.8);  // Full velocity
                allowed
80          move_group.setMaxAccelerationScalingFactor(0.8);  // Full
                acceleration allowed
81          move_group.setGoalTolerance(0.03);  // Position tolerance in
                meters
82          move_group.setSupportSurfaceName("table1");  // Default support
                 surface
83          move_group.allowReplanning(true);  // Allow replanning if
                initial plan fails
84          move_group.setNumPlanningAttempts(10);  // Number of planning
                retries
85
86          // Initialize Gazebo attach/detach service clients
87          attach_client = nh.serviceClient<gazebo_ros_link_attacher::
                Attach>("/link_attacher_node/attach");
88          detach_client = nh.serviceClient<gazebo_ros_link_attacher::
                Attach>("/link_attacher_node/detach");
89
90          // Wait for Gazebo services to become available
91          if (!attach_client.waitForExistence(ros::Duration(20.0)))
92              ROS_ERROR("Attach service not available!");
93          if (!detach_client.waitForExistence(ros::Duration(20.0)))
94              ROS_ERROR("Detach service not available!");
95
96          // Subscribe to laser scan topic for obstacle detection
```

```
97          sub = nh.subscribe<sensor_msgs::LaserScan>(
98              "/scan",
99              10,  // Queue size
100             &PickAndPlace::scanCallback,
101             this
102         );
103
104         // Subscribe to depth camera point cloud topic
105         sub1 = nh.subscribe(
106             "/rgbd_camera_depth/depth/points",
107             10,  // Queue size
108             &PickAndPlace::cloudCallback,
109             this
110         );
111     }
112
113     /**
114      * @brief Callback function for processing laser scan data to
            detect obstacles
115      *        on the left and right sides, adjusting robot speed or
            stopping accordingly.
116      * @param scan Pointer to the received LaserScan message.
117      */
118     void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
119     {
120         double angle_min = scan->angle_min;  // Minimum scan angle in
            radians
121         double angle_increment = scan->angle_increment;  // Angular
            resolution
122
123         std::vector<float> right_ranges;  // Ranges for right side
            detection
124         std::vector<float> left_ranges;   // Ranges for left side
            detection
125
126         // Iterate through scan ranges and classify by angle
127         for (size_t i = 0; i < scan->ranges.size(); ++i)
128         {
129             double angle_rad = angle_min + i * angle_increment;
```

```
130             double angle_deg = angle_rad * 180.0 / M_PI;  // Convert to
                    degrees
131
132             float range = scan->ranges[i];
133             if (std::isinf(range)) range = 0.0f;  // Handle infinite
                    ranges (no obstacle)
134
135             // Right side: angles between -20   and -1
136             if (angle_deg > -20.0 && angle_deg < -1.0)
137             {
138                 right_ranges.push_back(range);
139             }
140
141             // Left side: angles between 92   and 109
142             if (angle_deg > 92.0 && angle_deg < 109.0)
143             {
144                 left_ranges.push_back(range);
145             }
146         }
147
148         // Compute mean range for right side and check for person/
                obstacle
149         if (!right_ranges.empty())
150         {
151             float right_mean = std::accumulate(right_ranges.begin(),
                    right_ranges.end(), 0.0f) / right_ranges.size();
152             if (right_mean > 1.0 && right_mean < 2.0)  // Threshold for
                    stopping
153             {
154                 // ROS_INFO("Detecting a person on the right, mean
                        range: %.2f, stop the robot!", right_mean);
155                 stop = true;
156             }
157             else
158             {
159                 stop = false;
160             }
161         }
162
```

```
163         // Compute mean range for left side and check for person/
                obstacle
164         if (!left_ranges.empty())
165         {
166             float left_mean = std::accumulate(left_ranges.begin(),
                    left_ranges.end(), 0.0f) / left_ranges.size();
167             if (left_mean >= 0.9 && left_mean < 2.0)  // Threshold for
                    slowing down
168             {
169                 // ROS_INFO("Detecting a person on the left, mean range
                        : %.2f, slow down the robot!", left_mean);
170                 slow_down = true;
171             }
172             else
173             {
174                 slow_down = false;
175             }
176         }
177 
178         // Adjust robot motion based on detection
179         if (stop)
180         {
181             // move_group.setMaxVelocityScalingFactor(0.01);  //
                    Minimal velocity
182             move_group.stop();  // Immediate stop
183             ROS_INFO("Stopping the robot...");
184         }
185         else if (slow_down)
186         {
187             move_group.setMaxVelocityScalingFactor(0.2);  // Reduced
                    velocity
188             ROS_INFO("Slowing down the robot...");
189         }
190         else
191         {
192             move_group.setMaxVelocityScalingFactor(0.8);  // Normal
                    velocity
193             // ROS_INFO("No obstacles detected, robot can proceed.");
194         }
```

```
195     }
196
197     /**
198      * @brief Saves the received point cloud to a PCD file and triggers
             processing.
199      * @param pick_and_place Reference to the PickAndPlace instance.
200      */
201     void save(PickAndPlace& pick_and_place)
202     {
203         ros::Rate rate(10);  // Loop rate at 10 Hz
204         ROS_INFO("Waiting for point cloud...");
205
206         // Wait until point cloud is saved by callback
207         while (ros::ok())
208         {
209             if (has_saved)
210             {
211                 ROS_INFO("Point cloud saved. Starting processing...");
212                 break;
213             }
214             else
215             {
216                 ROS_INFO("Waiting for point cloud to be saved...");
217                 ros::spinOnce();  // Process callbacks
218             }
219         }
220
221         // Define input and output paths for PCD files
222         std::string path_input = "/home/adrian/catkin_ws/src/3D_cv/src/
               inputs/";
223         std::string path_output = "/home/adrian/catkin_ws/src/3D_cv/src
               /outputs/";
224         std::string input_pcd = path_input + "test2.pcd";
225         std::string output_pcd = path_output + "filtered_points.pcd";
226
227         // Process for cylinder detection and hole detection
228         pick_and_place.process_cylinder(input_pcd, output_pcd,
               tf_buffer);
229         pick_and_place.process_hole(input_pcd, output_pcd, tf_buffer);
```

```
230         // ros::spinOnce();
231         // rate.sleep();
232     }
233 
234     /**
235      * @brief Callback for saving the first received point cloud from
             depth camera.
236      * @param msg Pointer to the PointCloud2 message.
237      */
238     void cloudCallback(const sensor_msgs::PointCloud2ConstPtr& msg)
239     {
240         if (has_saved) return;  // Only save once
241 
242         pcl::PointCloud<pcl::PointXYZRGB> cloud;  // RGB point cloud
243         pcl::fromROSMsg(*msg, cloud);  // Convert ROS message to PCL
244 
245         std::string filename = "/home/adrian/catkin_ws/src/3D_cv/src/
               inputs/test2.pcd";
246         if (pcl::io::savePCDFileASCII(filename, cloud) == 0)  // Save
               as ASCII PCD
247         {
248             ROS_INFO("Saved single point cloud to %s", filename.c_str()
                   );
249             has_saved = true;  // Flag to prevent multiple saves
250         }
251         else
252         {
253             ROS_ERROR("Failed to save point cloud.");
254         }
255     }
256 
257     /**
258      * @brief Processes the point cloud to detect and segment cylinders
             (pegs).
259      *        Applies filtering, plane removal, normal estimation, and
            RANSAC segmentation.
260      *        Computes centroids in world frame and stores them.
261      * @param input_pcd Path to input PCD file.
262      * @param output_pcd Path to output PCD file for segmented
```

```
        cylinders.
263      * @param tf_buffer TF buffer for transformations.
264      */
265     void process_cylinder(const std::string& input_pcd, const std::
            string& output_pcd, tf2_ros::Buffer& tf_buffer)
266     {
267         // Load point cloud from PCD file
268         pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<
                pcl::PointXYZ>);
269         pcl::PCDReader cloud_reader;
270         cloud_reader.read(input_pcd, *cloud);
271 
272         pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered(new pcl::
                PointCloud<pcl::PointXYZ>);
273 
274         // Step 1: Voxel grid downsampling to reduce point density
275         pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
276         voxel_filter.setInputCloud(cloud);
277         voxel_filter.setLeafSize(0.001f, 0.001f, 0.001f);  // Voxel
                size in meters (tunable)
278         voxel_filter.filter(*cloud_filtered);
279 
280         // Step 2: Plane segmentation and removal using RANSAC
281         pcl::SACSegmentation<pcl::PointXYZ> seg;
282         pcl::PointIndices::Ptr inliers_plane(new pcl::PointIndices);
283         pcl::ModelCoefficients::Ptr coefficients_plane(new pcl::
                ModelCoefficients);
284         pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane(new pcl::
                PointCloud<pcl::PointXYZ>);
285 
286         seg.setOptimizeCoefficients(true);  // Optimize plane
                coefficients
287         seg.setModelType(pcl::SACMODEL_PLANE);
288         seg.setMethodType(pcl::SAC_RANSAC);
289         seg.setMaxIterations(200);  // RANSAC iterations
290         seg.setDistanceThreshold(0.01);  // Inlier distance threshold
291         seg.setInputCloud(cloud_filtered);
292         seg.segment(*inliers_plane, *coefficients_plane);
293 
```

```
294         // Extract non-plane points (negative indices)
295         pcl::ExtractIndices<pcl::PointXYZ> neg_plane_extracted;
296         neg_plane_extracted.setInputCloud(cloud_filtered);
297         neg_plane_extracted.setIndices(inliers_plane);
298         neg_plane_extracted.setNegative(true);  // Keep everything
                except plane
299         neg_plane_extracted.filter(*cloud_filtered);
300 
301         // Step 3: Pass-through filter to crop along Z-axis
302         pcl::PassThrough<pcl::PointXYZ> pass;
303         pass.setInputCloud(cloud_filtered);
304         pass.setFilterFieldName("z");
305         pass.setFilterLimits(0.0, 1.65);  // Z-range limits (tunable)
306         pass.filter(*cloud_filtered);
307 
308         // Step 4: Statistical outlier removal to denoise
309         pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
310         sor.setInputCloud(cloud_filtered);
311         sor.setMeanK(50);  // Number of neighbors for mean distance
312         sor.setStddevMulThresh(1.0);  // Standard deviation threshold
313         sor.filter(*cloud_filtered);
314 
315         // Step 5: Normal estimation for surface orientation
316         pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals(new pcl::
                PointCloud<pcl::PointNormal>);
317         pcl::NormalEstimation<pcl::PointXYZ, pcl::PointNormal>
                normal_estimator;
318         pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::
                KdTree<pcl::PointXYZ>);
319         normal_estimator.setSearchMethod(tree);
320         normal_estimator.setInputCloud(cloud_filtered);
321         normal_estimator.setKSearch(30);  // Neighbors for normal
                computation (tunable)
322         normal_estimator.compute(*cloud_normals);
323 
324         // Step 6: Cylinder segmentation using RANSAC on normals
325         pcl::SACSegmentationFromNormals<pcl::PointXYZ, pcl::PointNormal
                > cylinder_segmentation;
326         cylinder_segmentation.setOptimizeCoefficients(true);
```

```
327         cylinder_segmentation.setModelType(pcl::SACMODEL_CYLINDER);
328         cylinder_segmentation.setMethodType(pcl::SAC_RANSAC);
329         cylinder_segmentation.setNormalDistanceWeight(0.05);  // Weight
                for normal distance
330         cylinder_segmentation.setMaxIterations(10000);  // High
               iterations for accuracy
331         cylinder_segmentation.setDistanceThreshold(0.005);  // Inlier
               threshold (tunable)
332         cylinder_segmentation.setRadiusLimits(0.016, 0.022);  //
               Cylinder radius range (tunable)
333
334         // Variables for inliers, coefficients, and extracted cylinders
335         pcl::PointIndices::Ptr inliers_cylinder(new pcl::PointIndices);
336         pcl::ModelCoefficients::Ptr coefficients_cylinder(new pcl::
               ModelCoefficients);
337         pcl::ExtractIndices<pcl::PointXYZ> cylinder_extracted;
338         pcl::ExtractIndices<pcl::PointNormal>
               cylinder_indices_extractor_temp;
339         pcl::PointCloud<pcl::PointXYZ>::Ptr all_cylinders(new pcl::
               PointCloud<pcl::PointXYZ>);
340
341         int count = 0;  // Counter for detected cylinders
342         int l = 0;      // Counter for valid cylinders
343         while (true)  // Loop to segment multiple cylinders
344         {
345             pcl::PointCloud<pcl::PointXYZ>::Ptr cylinder_cloud(new pcl
                   ::PointCloud<pcl::PointXYZ>);
346
347             // Perform segmentation
348             cylinder_segmentation.setInputCloud(cloud_filtered);
349             cylinder_segmentation.setInputNormals(cloud_normals);
350             cylinder_segmentation.segment(*inliers_cylinder, *
                   coefficients_cylinder);
351
352             // Extract cylinder points
353             cylinder_extracted.setInputCloud(cloud_filtered);
354             cylinder_extracted.setIndices(inliers_cylinder);
355             cylinder_extracted.setNegative(false);  // Keep inliers
356             cylinder_extracted.filter(*cylinder_cloud);
```

```
357
358             if (!cylinder_cloud->points.empty())  // If cylinder found
359             {
360                 // Compute centroid of the cylinder
361                 Eigen::Vector4f centroid;
362                 pcl::compute3DCentroid(*cylinder_cloud, centroid);
363
364                 // Transform centroid from camera frame to world frame
365                 geometry_msgs::PointStamped point_cam;
366                 point_cam.header.stamp = ros::Time(0);  // Latest
                        available transform
367                 point_cam.header.frame_id = "
                        rgbd_camera_depth_optical_frame";  // Camera frame
368                 point_cam.point.x = centroid[0];
369                 point_cam.point.y = centroid[1];
370                 point_cam.point.z = centroid[2];
371
372                 geometry_msgs::PointStamped point_world;
373                 try {
374                     point_world = tf_buffer.transform(point_cam, "world
                            ", ros::Duration(1.0));
375
376                     std::cout << "Centroid in world frame: ("
377                               << point_world.point.x << ", "
378                               << point_world.point.y << ", "
379                               << point_world.point.z << ")" << std::
                                      endl;
380                 } catch (tf2::TransformException& ex) {
381                     ROS_WARN("TF transform failed: %s", ex.what());
382                 }
383
384                 // Store world-frame centroid
385                 centroid_cylinder[0] = point_world.point.x;
386                 centroid_cylinder[1] = point_world.point.y;
387                 centroid_cylinder[2] = point_world.point.z;
388
389                 // Accumulate large cylinders (filter by point count)
390                 if (cylinder_cloud->points.size() > 90) {
391                     *all_cylinders += *cylinder_cloud;
```

```
392                     l++;
393                 }
394 
395                 // Remove segmented cylinder from input for next
                        iteration
396                 cylinder_extracted.setNegative(true);
397                 cylinder_extracted.filter(*cloud_filtered);
398 
399                 // Update normals by removing segmented points
400                 cylinder_indices_extractor_temp.setInputCloud(
                        cloud_normals);
401                 cylinder_indices_extractor_temp.setIndices(
                        inliers_cylinder);
402                 cylinder_indices_extractor_temp.setNegative(true);
403                 cylinder_indices_extractor_temp.filter(*cloud_normals);
404 
405                 if (count >= 1)  // Limit to one cylinder for this
                        application
406                 {
407                     break;
408                 }
409                 count++;
410             }
411             else {
412                 std::cout << "No more cylinders. Terminating
                        segmentation." << std::endl;
413                 break;
414             }
415         }
416 
417         // Save segmented cylinders to output PCD
418         pcl::PCDWriter cloud_writer;
419         cloud_writer.write<pcl::PointXYZ>(output_pcd, *all_cylinders,
                false);
420         // cloud_writer.write<pcl::PointXYZ>(output_pcd, *
                cloud_filtered, false);
421     }
422 
423     /**
```

```
424      * @brief Processes the point cloud to detect holes (circular
             features on surfaces).
425      *        Involves clustering, top surface extraction, and 2D
             circle segmentation.
426      *        Filters by height and position, stores centroid in camera
              frame.
427      * @param input_pcd Path to input PCD file.
428      * @param output_pcd Path to output PCD file for segmented circles.
429      * @param tf_buffer TF buffer for transformations (used for world
             transform).
430      */
431     void process_hole(const std::string& input_pcd, const std::string&
            output_pcd, tf2_ros::Buffer& tf_buffer)
432     {
433         // Load and filter point cloud (similar to cylinder processing)
434         pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<
                pcl::PointXYZ>);
435         pcl::PCDReader cloud_reader;
436         cloud_reader.read(input_pcd, *cloud);
437 
438         pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered(new pcl::
                PointCloud<pcl::PointXYZ>);
439 
440         // 1. Voxel downsampling
441         pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
442         voxel_filter.setInputCloud(cloud);
443         voxel_filter.setLeafSize(0.003f, 0.003f, 0.003f);  // Slightly
                larger voxels
444         voxel_filter.filter(*cloud_filtered);
445 
446         // 2. Remove dominant plane
447         pcl::SACSegmentation<pcl::PointXYZ> seg_plane;
448         pcl::PointIndices::Ptr inliers_plane(new pcl::PointIndices);
449         pcl::ModelCoefficients::Ptr coefficients_plane(new pcl::
                ModelCoefficients);
450 
451         seg_plane.setOptimizeCoefficients(true);
452         seg_plane.setModelType(pcl::SACMODEL_PLANE);
453         seg_plane.setMethodType(pcl::SAC_RANSAC);
```

```
454         seg_plane.setMaxIterations(200);
455         seg_plane.setDistanceThreshold(0.01);
456         seg_plane.setInputCloud(cloud_filtered);
457         seg_plane.segment(*inliers_plane, *coefficients_plane);
458 
459         pcl::ExtractIndices<pcl::PointXYZ> extract;
460         extract.setInputCloud(cloud_filtered);
461         extract.setIndices(inliers_plane);
462         extract.setNegative(true);
463         extract.filter(*cloud_filtered);
464 
465         // 3. Z-axis pass-through filter
466         pcl::PassThrough<pcl::PointXYZ> pass;
467         pass.setInputCloud(cloud_filtered);
468         pass.setFilterFieldName("z");
469         pass.setFilterLimits(0.0, 1.65);
470         pass.filter(*cloud_filtered);
471 
472         // 4. Outlier removal
473         pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
474         sor.setInputCloud(cloud_filtered);
475         sor.setMeanK(50);
476         sor.setStddevMulThresh(1.0);
477         sor.filter(*cloud_filtered);
478 
479         // Transform cloud to world frame for clustering
480         pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_world(new pcl::
               PointCloud<pcl::PointXYZ>);
481         geometry_msgs::TransformStamped transform_stamped;
482 
483         try {
484             transform_stamped = tf_buffer.lookupTransform(
485                 "world",  // Target frame
486                 "rgbd_camera_depth_optical_frame",  // Source frame (
                       fix: set frame_id)
487                 ros::Time(0),
488                 ros::Duration(1.0)
489             );
490 
```

```
491             Eigen::Matrix4f transform;
492             pcl_ros::transformAsMatrix(transform_stamped.transform,
                    transform);
493             pcl::transformPointCloud(*cloud_filtered, *cloud_world,
                    transform);
494
495         } catch (tf2::TransformException &ex) {
496             ROS_WARN("Transform failed: %s", ex.what());
497             return;
498         }
499
500         // 5. Euclidean clustering to separate objects
501         pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::
                KdTree<pcl::PointXYZ>);
502         tree->setInputCloud(cloud_filtered);  // Note: Using filtered
                for tree, but world for clustering
503
504         std::vector<pcl::PointIndices> cluster_indices;
505         pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
506         ec.setClusterTolerance(0.01);  // 1 cm tolerance
507         ec.setMinClusterSize(50);      // Minimum cluster size
508         ec.setMaxClusterSize(50000);   // Maximum cluster size
509         ec.setSearchMethod(tree);
510         ec.setInputCloud(cloud_world);  // Cluster in world frame
511         ec.extract(cluster_indices);
512
513         // Extract top surfaces from clusters (points near max Z)
514         pcl::PointCloud<pcl::PointXYZ>::Ptr top_surfaces(new pcl::
                PointCloud<pcl::PointXYZ>);
515
516         for (const auto& indices : cluster_indices) {
517             // Find max Z in cluster
518             float max_z = -std::numeric_limits<float>::max();
519             for (int idx : indices.indices) {
520                 if (cloud_world->points[idx].z > max_z)
521                     max_z = cloud_world->points[idx].z;
522             }
523
524             // Keep points within 2 cm of max Z (top surface)
```

```
            for (int idx : indices.indices) {
                if (std::abs(cloud_world->points[idx].z - max_z) <=
                    0.02f) {
                    top_surfaces->points.push_back(cloud_world->points[
                        idx]);
                }
            }
        }

        top_surfaces->width = top_surfaces->points.size();
        top_surfaces->height = 1;
        top_surfaces->is_dense = true;

        // Normal estimation on top surfaces
        pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals(new pcl::
            PointCloud<pcl::PointNormal>);
        pcl::NormalEstimation<pcl::PointXYZ, pcl::PointNormal>
            normal_estimator;
        normal_estimator.setSearchMethod(tree);
        normal_estimator.setInputCloud(top_surfaces);
        normal_estimator.setKSearch(30);  // Tunable
        normal_estimator.compute(*cloud_normals);

        // 2D Circle segmentation on top surfaces
        pcl::SACSegmentationFromNormals<pcl::PointXYZ, pcl::PointNormal
            > circle_segmentation;
        circle_segmentation.setOptimizeCoefficients(true);
        circle_segmentation.setModelType(pcl::SACMODEL_CIRCLE2D);  // 2
            D circle model
        circle_segmentation.setMethodType(pcl::SAC_RANSAC);
        circle_segmentation.setNormalDistanceWeight(0.05);
        circle_segmentation.setMaxIterations(10000);
        circle_segmentation.setDistanceThreshold(0.005);  // Tunable
        circle_segmentation.setRadiusLimits(0.018, 0.03);  // Circle
            radius range (tunable)

        // Variables for circle extraction
        pcl::PointIndices::Ptr inliers_circle(new pcl::PointIndices);
        pcl::ModelCoefficients::Ptr coefficients_circle(new pcl::
```

```
        ModelCoefficients);
    pcl::ExtractIndices<pcl::PointXYZ> circle_extracted;
    pcl::ExtractIndices<pcl::PointNormal>
        circle_indices_extractor_temp;
    pcl::PointCloud<pcl::PointXYZ>::Ptr all_circle(new pcl::
        PointCloud<pcl::PointXYZ>);

    int count = 0;
    int l = 0;
    while (true)  // Loop for multiple circles
    {
        pcl::PointCloud<pcl::PointXYZ>::Ptr circle_cloud(new pcl::
            PointCloud<pcl::PointXYZ>);

        // Segment circle
        circle_segmentation.setInputCloud(top_surfaces);
        circle_segmentation.setInputNormals(cloud_normals);
        circle_segmentation.segment(*inliers_circle, *
            coefficients_circle);

        // Extract circle points
        circle_extracted.setInputCloud(top_surfaces);
        circle_extracted.setIndices(inliers_circle);
        circle_extracted.setNegative(false);
        circle_extracted.filter(*circle_cloud);

        if (!circle_cloud->points.empty()) {
            // Compute centroid
            Eigen::Vector4f centroid;
            pcl::compute3DCentroid(*circle_cloud, centroid);

            // Prepare for TF transform (camera frame)
            geometry_msgs::PointStamped point_cam;
            point_cam.header.stamp = ros::Time(0);
            point_cam.header.frame_id = "
                rgbd_camera_depth_optical_frame";
            point_cam.point.x = centroid[0];
            point_cam.point.y = centroid[1];
            point_cam.point.z = centroid[2];
```

```
590
591                 // Filter by Z-height (skip high centroids)
592                 if (centroid[2] > 1.2) {
593                     circle_extracted.setNegative(true);
594                     circle_extracted.filter(*top_surfaces);
595
596                     circle_indices_extractor_temp.setInputCloud(
                            cloud_normals);
597                     circle_indices_extractor_temp.setIndices(
                            inliers_circle);
598                     circle_indices_extractor_temp.setNegative(true);
599                     circle_indices_extractor_temp.filter(*cloud_normals
                            );
600                     continue;
601                 }
602
603                 // Filter by X-position (skip low centroids)
604                 if (centroid[0] < 0.47) {
605                     circle_extracted.setNegative(true);
606                     circle_extracted.filter(*top_surfaces);
607
608                     circle_indices_extractor_temp.setInputCloud(
                            cloud_normals);
609                     circle_indices_extractor_temp.setIndices(
                            inliers_circle);
610                     circle_indices_extractor_temp.setNegative(true);
611                     circle_indices_extractor_temp.filter(*cloud_normals
                            );
612                     continue;
613                 }
614
615                 // Log centroid in camera frame
616                 std::cout << "Centroid of the circle in the camera
                        frame: ("
617                           << centroid[0] << ", "
618                           << centroid[1] << ","
619                           << centroid[2] << ")" << std::endl;
620
621                 // Store centroid (camera frame for hole)
```

```
622                 centroid_hole[0] = centroid[0];
623                 centroid_hole[1] = centroid[1];
624                 centroid_hole[2] = centroid[2];
625
626                 // Accumulate large circles
627                 if (circle_cloud->points.size() > 90) {
628                     *all_circle += *circle_cloud;
629                     l++;
630                 }
631
632                 // Remove segmented circle
633                 circle_extracted.setNegative(true);
634                 circle_extracted.filter(*top_surfaces);
635
636                 // Update normals
637                 circle_indices_extractor_temp.setInputCloud(
                        cloud_normals);
638                 circle_indices_extractor_temp.setIndices(inliers_circle
                        );
639                 circle_indices_extractor_temp.setNegative(true);
640                 circle_indices_extractor_temp.filter(*cloud_normals);
641             }
642             else {
643                 std::cout << "No more circles. Terminating segmentation
                        ." << std::endl;
644                 break;
645                 got_centroid = true;  // Flag for successful detection
646             }
647         }
648
649         // Save segmented circles
650         pcl::PCDWriter cloud_writer;
651         cloud_writer.write<pcl::PointXYZ>(output_pcd, *all_circle,
                false);
652         // ROS_INFO(" Saved %d detected pegs to %s", count, output_pcd.
                c_str());
653     }
654
655     /**
```

```
656         * @brief Sets gripper posture to open position.
657         * @param posture Joint trajectory for gripper.
658         */
659        void openGripper(trajectory_msgs::JointTrajectory& posture)
660        {
661            posture.joint_names.resize(1);
662            posture.joint_names[0] = "finger_joint";  // Gripper joint name
663            posture.points.resize(1);
664            posture.points[0].positions.resize(1);
665            posture.points[0].positions[0] = 0;  // Open position (0
                radians)
666            posture.points[0].time_from_start = ros::Duration(1.0);  //
                Execution time
667            ROS_INFO("Opening gripper: position = %f", posture.points[0].
                positions[0]);
668        }
669
670        /**
671         * @brief Sets gripper posture to closed position.
672         * @param posture Joint trajectory for gripper.
673         */
674        void closedGripper(trajectory_msgs::JointTrajectory& posture)
675        {
676            posture.joint_names.resize(1);
677            posture.joint_names[0] = "finger_joint";
678            posture.points.resize(1);
679            posture.points[0].positions.resize(1);
680            posture.points[0].positions[0] = 0.3;  // Close position (
                tunable, ~0.3 radians)
681            posture.points[0].time_from_start = ros::Duration(1.0);
682            ROS_INFO("Closing gripper: position = %f", posture.points[0].
                positions[0]);
683        }
684
685        /**
686         * @brief Performs pick operation using MoveIt grasps at computed
                cylinder centroid.
687         * @param move_group Reference to MoveGroupInterface for planning.
688         */
```

```
689     void ur5_pick(moveit::planning_interface::MoveGroupInterface&
            move_group)
690     {
691         std::vector<moveit_msgs::Grasp> grasps;
692         grasps.resize(1);
693 
694         // Define grasp pose in world frame
695         grasps[0].grasp_pose.header.frame_id = "world";
696         tf2::Quaternion orientation;
697         orientation.setRPY(M_PI, 0, M_PI_2);  // Roll-Pitch-Yaw
                orientation for grasp
698         grasps[0].grasp_pose.pose.orientation = tf2::toMsg(orientation)
                ;
699         grasps[0].grasp_pose.pose.position.x = centroid_cylinder[0] +
                0.003;  // Offset for grasp
700         grasps[0].grasp_pose.pose.position.y = centroid_cylinder[1] -
                0.032;
701         grasps[0].grasp_pose.pose.position.z = 1.32;  // Fixed Z height
                 (tunable)
702         ROS_INFO("Grasp pose: x=%f, y=%f, z=%f", grasps[0].grasp_pose.
                pose.position.x,
703                  grasps[0].grasp_pose.pose.position.y, grasps[0].
                     grasp_pose.pose.position.z);
704 
705         // Pre-grasp approach direction and distance
706         grasps[0].pre_grasp_approach.direction.header.frame_id = "world
                ";
707         grasps[0].pre_grasp_approach.direction.vector.z = -1.0;  //
                Approach from above
708         grasps[0].pre_grasp_approach.min_distance = 0.12;  // Min
                approach distance
709         grasps[0].pre_grasp_approach.desired_distance = 0.15;  //
                Desired approach
710 
711         // Set pre- and post-grasp postures
712         openGripper(grasps[0].pre_grasp_posture);    // Open before
                grasp
713         closedGripper(grasps[0].grasp_posture);      // Close during
                grasp
```

```
714
715         move_group.setSupportSurfaceName("table1");
716
717         // Execute pick
718         if (!move_group.pick("cs1", grasps))  // Pick object ID "cs1"
719         {
720             ROS_ERROR("Pick operation failed!");
721             last_operation_failed = true;
722         }
723         else {
724             attachObject();  // Attach in Gazebo after successful pick
725             last_operation_failed = false;
726         }
727     }
728
729     /**
730      * @brief Attaches the picked collision object to the robot's tool
            in MoveIt scene.
731      */
732     void attachCollisionObject()
733     {
734         moveit_msgs::AttachedCollisionObject attached_object;
735         attached_object.link_name = "tool0";  // End-effector link
736         attached_object.object = collision_objects[2];  // "cs1"
            cylinder
737         attached_object.object.operation = moveit_msgs::CollisionObject
            ::ADD;
738         attached_object.touch_links = {
739             "robotiq_85_right_finger_tip_link",
740             "robotiq_85_left_finger_tip_link",
741             "robotiq_85_right_inner_knuckle_link",
742             "robotiq_85_left_inner_knuckle_link",
743             "wrist_3_link",
744             "tool0",
745             "robotiq_85_right_inner_knuckle_link",  // Duplicates for
                completeness
746             "robotiq_85_left_inner_knuckle_link",
747             "robotiq_85_right_finger_link",
748             "robotiq_85_left_finger_link",
```

```
749             "robotiq_85_right_finger_tip_link",
750             "robotiq_85_left_finger_tip_link"
751         };
752         planning_scene_interface.applyAttachedCollisionObject(
                attached_object);
753         ROS_INFO("Applied attached collision object: cs1 to tool0");
754     }
755
756     /**
757      * @brief Detaches the collision object from the robot's tool in
            MoveIt scene.
758      */
759     void detachCollisionObject()
760     {
761         moveit_msgs::AttachedCollisionObject attached_object;
762         attached_object.link_name = "tool0";
763         attached_object.object.operation = moveit_msgs::CollisionObject
                ::REMOVE;
764         attached_object.object.id = "cs1";  // Object ID to detach
765         planning_scene_interface.applyAttachedCollisionObject(
                attached_object);
766         ROS_INFO("Detached collision object: cs1 from tool0");
767     }
768
769     /**
770      * @brief Performs place operation using MoveIt at computed hole
            centroid.
771      * @param group Reference to MoveGroupInterface for planning.
772      */
773     void ur5_place(moveit::planning_interface::MoveGroupInterface&
            group)
774     {
775         std::vector<moveit_msgs::PlaceLocation> place_location;
776         place_location.resize(1);
777
778         // Define place pose in world frame
779         place_location[0].place_pose.header.frame_id = "world";
780         tf2::Quaternion orientation;
781         orientation.setRPY(0, 0, 0);  // Neutral orientation for place
```

```
782         place_location[0].place_pose.pose.orientation = tf2::toMsg(
                orientation);
783         place_location[0].place_pose.pose.position.x = centroid_hole
                [0];  // Use hole centroid
784         place_location[0].place_pose.pose.position.y = centroid_hole[1]
                 - 0.045;
785         place_location[0].place_pose.pose.position.z = 1.18;  // Fixed
                Z for placement
786         ROS_INFO("Place pose: x=%f, y=%f, z=%f", place_location[0].
                place_pose.pose.position.x,
787                  place_location[0].place_pose.pose.position.y,
                         place_location[0].place_pose.pose.position.z);
788
789         // Pre-place approach
790         place_location[0].pre_place_approach.direction.header.frame_id
                = "world";
791         place_location[0].pre_place_approach.direction.vector.z = -1.0;
                  // From above
792         place_location[0].pre_place_approach.min_distance = 0.08;
793         place_location[0].pre_place_approach.desired_distance = 0.18;
794
795         // Post-place posture (open gripper)
796         openGripper(place_location[0].post_place_posture);
797
798         group.setSupportSurfaceName("table2");  // Place on table2
799         group.allowReplanning(true);
800         group.setGoalTolerance(0.03);
801
802         // Execute place
803         if (!group.place("cs1", place_location)) {
804             ROS_ERROR("Place operation failed!");
805             last_operation_failed = true;
806         }
807         else {
808             detachObject();  // Detach in Gazebo after place
809             last_operation_failed = false;
810         }
811     }
812
```

```
813     /**
814      * @brief Attaches object in Gazebo simulation using service call.
815      */
816     void attachObject()
817     {
818         gazebo_ros_link_attacher::Attach srv;
819         srv.request.model_name_1 = "robot";        // Robot model
820         srv.request.link_name_1 = "wrist_3_link"; // Attachment link on
                robot
821         srv.request.model_name_2 = "blue_cylinder1";  // Object model
822         srv.request.link_name_2 = "base_link";     // Link on object
823 
824         if (attach_client.call(srv) && srv.response.ok)
825             ROS_INFO("Object attached successfully: %s::%s to %s::%s",
826                      srv.request.model_name_1.c_str(), srv.request.
                            link_name_1.c_str(),
827                      srv.request.model_name_2.c_str(), srv.request.
                            link_name_2.c_str());
828         else
829             ROS_ERROR("Failed to attach object: %s::%s to %s::%s",
830                       srv.request.model_name_1.c_str(), srv.request.
                             link_name_1.c_str(),
831                       srv.request.model_name_2.c_str(), srv.request.
                             link_name_2.c_str());
832     }
833 
834     /**
835      * @brief Detaches object in Gazebo simulation using service call.
836      */
837     void detachObject()
838     {
839         gazebo_ros_link_attacher::Attach srv;
840         srv.request.model_name_1 = "robot";
841         srv.request.link_name_1 = "wrist_3_link";
842         srv.request.model_name_2 = "blue_cylinder1";
843         srv.request.link_name_2 = "base_link";
844 
845         if (detach_client.call(srv) && srv.response.ok)
846             ROS_INFO("Object detached successfully: %s::%s from %s::%s"
```

```
                ,
847                     srv.request.model_name_1.c_str(), srv.request.
                            link_name_1.c_str(),
848                     srv.request.model_name_2.c_str(), srv.request.
                            link_name_2.c_str());
849         else
850             ROS_ERROR("Failed to detach object: %s::%s from %s::%s",
851                       srv.request.model_name_1.c_str(), srv.request.
                              link_name_1.c_str(),
852                       srv.request.model_name_2.c_str(), srv.request.
                              link_name_2.c_str());
853     }
854
855     /**
856      * @brief Adds static collision objects (tables, bins, initial
            cylinder) to MoveIt scene.
857      * @param planning_scene_interface Reference to
            PlanningSceneInterface.
858      */
859     void addCollisionObject(moveit::planning_interface::
           PlanningSceneInterface& planning_scene_interface)
860     {
861         collision_objects.resize(7);  // 7 objects: 2 tables, 1
               cylinder, 4 bins (base + 3)
862
863         // Table 1 (pick area)
864         collision_objects[0].id = "table1";
865         collision_objects[0].header.frame_id = "world";
866         collision_objects[0].primitives.resize(1);
867         collision_objects[0].primitives[0].type = shape_msgs::
               SolidPrimitive::BOX;
868         collision_objects[0].primitives[0].dimensions = {1.5, 0.8,
               1.015};  // Size: x,y,z
869         collision_objects[0].primitive_poses.resize(1);
870         collision_objects[0].primitive_poses[0].position.x = 0.6;
871         collision_objects[0].primitive_poses[0].position.y = -0.4;
872         collision_objects[0].primitive_poses[0].position.z = 0.505;
873         collision_objects[0].operation = moveit_msgs::CollisionObject::
               ADD;
```

```
874
875         // Table 2 (place area)
876         collision_objects[1].id = "table2";
877         collision_objects[1].header.frame_id = "world";
878         collision_objects[1].primitives.resize(1);
879         collision_objects[1].primitives[0].type = shape_msgs::
                SolidPrimitive::BOX;
880         collision_objects[1].primitives[0].dimensions = {1.5, 0.8,
                1.015};
881         collision_objects[1].primitive_poses.resize(1);
882         collision_objects[1].primitive_poses[0].position.x = 0.6;
883         collision_objects[1].primitive_poses[0].position.y = 0.402;
884         collision_objects[1].primitive_poses[0].position.z = 0.505;
885         collision_objects[1].operation = moveit_msgs::CollisionObject::
                ADD;
886
887         // Initial cylinder (cs1) on table1
888         collision_objects[2].id = "cs1";
889         collision_objects[2].header.frame_id = "world";
890         collision_objects[2].primitives.resize(1);
891         collision_objects[2].primitives[0].type = shape_msgs::
                SolidPrimitive::CYLINDER;
892         collision_objects[2].primitives[0].dimensions.resize(2);
893         collision_objects[2].primitives[0].dimensions[0] = 0.2;   //
                Height
894         collision_objects[2].primitives[0].dimensions[1] = 0.02;  //
                Radius
895         collision_objects[2].primitive_poses.resize(1);
896         collision_objects[2].primitive_poses[0].position.x = 0.3;
897         collision_objects[2].primitive_poses[0].position.y = 0.6;
898         collision_objects[2].primitive_poses[0].position.z = 1.12;
899         collision_objects[2].primitive_poses[0].orientation.w = 1.0;
900         collision_objects[2].operation = moveit_msgs::CollisionObject::
                ADD;
901
902         // Bin base
903         collision_objects[3].id = "bin_base";
904         collision_objects[3].header.frame_id = "world";
905         collision_objects[3].primitives.resize(1);
```

```
906         collision_objects[3].primitives[0].type = shape_msgs::
                SolidPrimitive::BOX;
907         collision_objects[3].primitives[0].dimensions = {0.5, 1.5,
                0.74};
908         collision_objects[3].primitive_poses.resize(1);
909         collision_objects[3].primitive_poses[0].position.x = -0.6;
910         collision_objects[3].primitive_poses[0].position.y = 0.0;
911         collision_objects[3].primitive_poses[0].position.z = 0.35;
912         collision_objects[3].operation = moveit_msgs::CollisionObject::
                ADD;
913
914         // Bin 1
915         collision_objects[4].id = "bin1";
916         collision_objects[4].header.frame_id = "world";
917         collision_objects[4].primitives.resize(1);
918         collision_objects[4].primitives[0].type = shape_msgs::
                SolidPrimitive::BOX;
919         collision_objects[4].primitives[0].dimensions = {0.4, 0.4,
                0.02};
920         collision_objects[4].primitive_poses.resize(1);
921         collision_objects[4].primitive_poses[0].position.x = -0.6;
922         collision_objects[4].primitive_poses[0].position.y = 0.5;
923         collision_objects[4].primitive_poses[0].position.z = 0.76;
924         collision_objects[4].operation = moveit_msgs::CollisionObject::
                ADD;
925
926         // Bin 2
927         collision_objects[5].id = "bin2";
928         collision_objects[5].header.frame_id = "world";
929         collision_objects[5].primitives.resize(1);
930         collision_objects[5].primitives[0].type = shape_msgs::
                SolidPrimitive::BOX;
931         collision_objects[5].primitives[0].dimensions = {0.4, 0.4,
                0.02};
932         collision_objects[5].primitive_poses.resize(1);
933         collision_objects[5].primitive_poses[0].position.x = -0.6;
934         collision_objects[5].primitive_poses[0].position.y = 0.0;
935         collision_objects[5].primitive_poses[0].position.z = 0.76;
936         collision_objects[5].operation = moveit_msgs::CollisionObject::
```

```
            ADD;
937
938         // Bin 3
939         collision_objects[6].id = "bin3";
940         collision_objects[6].header.frame_id = "world";
941         collision_objects[6].primitives.resize(1);
942         collision_objects[6].primitives[0].type = shape_msgs::
                SolidPrimitive::BOX;
943         collision_objects[6].primitives[0].dimensions = {0.4, 0.4,
                0.02};
944         collision_objects[6].primitive_poses.resize(1);
945         collision_objects[6].primitive_poses[0].position.x = -0.6;
946         collision_objects[6].primitive_poses[0].position.y = -0.5;
947         collision_objects[6].primitive_poses[0].position.z = 0.76;
948         collision_objects[6].operation = moveit_msgs::CollisionObject::
                ADD;
949
950         // Apply all collision objects to the planning scene
951         planning_scene_interface.applyCollisionObjects(
                collision_objects);
952         ROS_INFO("Added %zu collision objects to planning scene",
                collision_objects.size());
953     }
954
955     /**
956      * @brief Main execution loop for the pick-and-place sequence.
957      *        Handles steps: add objects, pick, attach, place, detach.
958      *        Integrates obstacle avoidance and retries on failure.
959      * @param pick_and_place Reference to PickAndPlace instance.
960      * @param group MoveGroup for motion.
961      * @param planning_scene_interface Planning scene.
962      */
963     void execution(PickAndPlace& pick_and_place,
964                    moveit::planning_interface::MoveGroupInterface&
                           group,
965                    moveit::planning_interface::PlanningSceneInterface&
                           planning_scene_interface)
966     {
967         // This function orchestrates the pick-and-place sequence with
```

```
        obstacle checks
968     ros::Rate rate(10);  // 10 Hz loop rate
969     int step = 1;  // Current execution step
970     while (ros::ok())
971     {
972         switch (step)
973         {
974             case 1:  // Step 1: Add collision objects to scene
975                 if (stop)  // Check for full stop
976                 {
977                     ROS_INFO("Step 1: Waiting for clear path");
978                     ros::Duration(2.0).sleep();
979                     break;  // Stay in step 1
980                 }
981                 ROS_INFO("Adding collision objects");
982                 pick_and_place.addCollisionObject(
                        planning_scene_interface);
983                 ros::Duration(2.0).sleep();  // Pause for scene
                        update
984                 step++;
985                 break;
986
987             case 2:  // Step 2: Pick operation
988                 if (stop)
989                 {
990                     ROS_INFO("Step 2: Waiting for clear path");
991                     ros::Duration(2.0).sleep();
992                     break;  // Retry step 2
993                 }
994                 // Adjust speed based on slowdown flag
995                 if (slow_down)
996                 {
997                     group.setMaxVelocityScalingFactor(0.4);
998                     group.setMaxAccelerationScalingFactor(0.4);
999                     // ROS_INFO("Applying slowdown: 40% speed");
1000                }
1001                else
1002                {
1003                    group.setMaxVelocityScalingFactor(1.0);
```

```
                group.setMaxAccelerationScalingFactor(1.0);
                // ROS_INFO("Applying default speed: 80% speed
                    ");
            }
            pick_and_place.ur5_pick(group);
            if (last_operation_failed)
            {
                ROS_WARN("Pick failed, retrying step 2");
                ros::Duration(2.0).sleep();
                break;  // Retry
            }
            step++;
            break;

        case 3:  // Step 3: Attach object to robot
            if (stop)
            {
                ROS_INFO("Step 3: Waiting for clear path");
                ros::Duration(2.0).sleep();
                break;  // Retry
            }
            pick_and_place.attachCollisionObject();
            ros::Duration(2.0).sleep();
            step++;
            break;

        case 4:  // Step 4: Place operation
            if (stop)
            {
                ROS_INFO("Step 4: Waiting for clear path");
                ros::Duration(2.0).sleep();
                break;  // Retry
            }
            // Adjust speed
            if (slow_down)
            {
                group.setMaxVelocityScalingFactor(0.4);
                group.setMaxAccelerationScalingFactor(0.4);
                // ROS_INFO("Applying slowdown: 40% speed");
```

```
                }
                else
                {
                    group.setMaxVelocityScalingFactor(1.0);
                    group.setMaxAccelerationScalingFactor(1.0);
                    // ROS_INFO("Applying default speed: 80% speed
                        ");
                }
                pick_and_place.ur5_place(group);
                if (last_operation_failed)
                {
                    ROS_WARN("Place failed, retrying step 4");
                    ros::Duration(2.0).sleep();
                    break;  // Retry
                }
                step++;
                break;

            case 5:  // Step 5: Detach object
                if (stop)
                {
                    ROS_INFO("Step 5: Waiting for clear path");
                    ros::Duration(2.0).sleep();
                    break;  // Retry
                }
                pick_and_place.detachCollisionObject();
                ros::Duration(2.0).sleep();
                step++;  // Complete
                break;

            default:
                ROS_INFO("Task completed");
                ros::shutdown();
                break;
        }

        rate.sleep();  // Maintain loop rate
        ros::spinOnce();  // Process callbacks
    }
```

```
1080        }
1081
1082    private:
1083        // MoveIt interfaces
1084        moveit::planning_interface::PlanningSceneInterface
                planning_scene_interface;
1085        moveit::planning_interface::MoveGroupInterface move_group;
1086        moveit::planning_interface::MoveGroupInterface gripper_group;
1087
1088        // Gazebo service clients
1089        ros::ServiceClient attach_client;
1090        ros::ServiceClient detach_client;
1091
1092        // Collision objects vector
1093        std::vector<moveit_msgs::CollisionObject> collision_objects;
1094
1095        // TF transformation buffer and listener
1096        tf2_ros::Buffer tf_buffer;
1097        tf2_ros::TransformListener tf_listener;  // Starts TF listener
                thread
1098
1099        // Flags for laser scan-based control
1100        bool stop = false;        // Full stop flag
1101        bool slow_down = false;   // Slowdown flag
1102
1103        // Point cloud processing flags
1104        bool has_saved = false;   // Point cloud saved flag
1105        bool got_centroid = false;   // Centroid detected flag
1106
1107        bool last_operation_failed = false;   // Track pick/place failures
1108
1109        // Subscribers
1110        ros::Subscriber sub;      // Laser scan subscriber
1111        ros::Subscriber sub1;     // Point cloud subscriber
1112
1113        // Stored centroids (cylinder in world, hole in camera)
1114        std::array<double, 3> centroid_cylinder;
1115        std::array<double, 3> centroid_hole;
1116    };
```

```
1117
1118 /**
1119  * @brief Main function initializes ROS, MoveIt, and runs the pick-and-
          place execution.
1120  * @param argc Argument count.
1121  * @param argv Argument vector.
1122  * @return Exit code.
1123  */
1124 int main(int argc, char** argv)
1125 {
1126     ros::init(argc, argv, "ur5_pick_and_place");  // Initialize ROS
          node
1127     ros::NodeHandle nh;
1128     // ros::Subscriber sub = nh.subscribe("/scan", 10, scanCallback);
          // Handled in class
1129     ros::AsyncSpinner spinner(1);  // Asynchronous spinner for
          callbacks
1130     spinner.start();
1131
1132     ros::WallDuration(2.0).sleep();  // Wait for system initialization
1133     moveit::planning_interface::PlanningSceneInterface
          planning_scene_interface;
1134     moveit::planning_interface::MoveGroupInterface group("
          ur5_manipulator");
1135
1136     // Configure group parameters
1137     group.setPlanningTime(40.0);  // Longer planning time
1138     group.setNumPlanningAttempts(10);
1139     group.setMaxVelocityScalingFactor(0.8);
1140     group.setMaxAccelerationScalingFactor(0.8.0);
1141     group.setGoalTolerance(0.03);
1142
1143     PickAndPlace pick_and_place(nh);  // Create instance
1144
1145     pick_and_place.save(pick_and_place);  // Save and process point
          cloud
1146
1147     ros::WallDuration(5.0).sleep();  // Wait for processing
1148
```

```
    // Execute the sequence
    pick_and_place.execution(pick_and_place, group,
        planning_scene_interface);

    ros::waitForShutdown();  // Wait for shutdown
    return 0;
}
```

Listing 1: Complete Pick and Place Implementation with Detailed Comments

Task Execution Snapshots and Video link

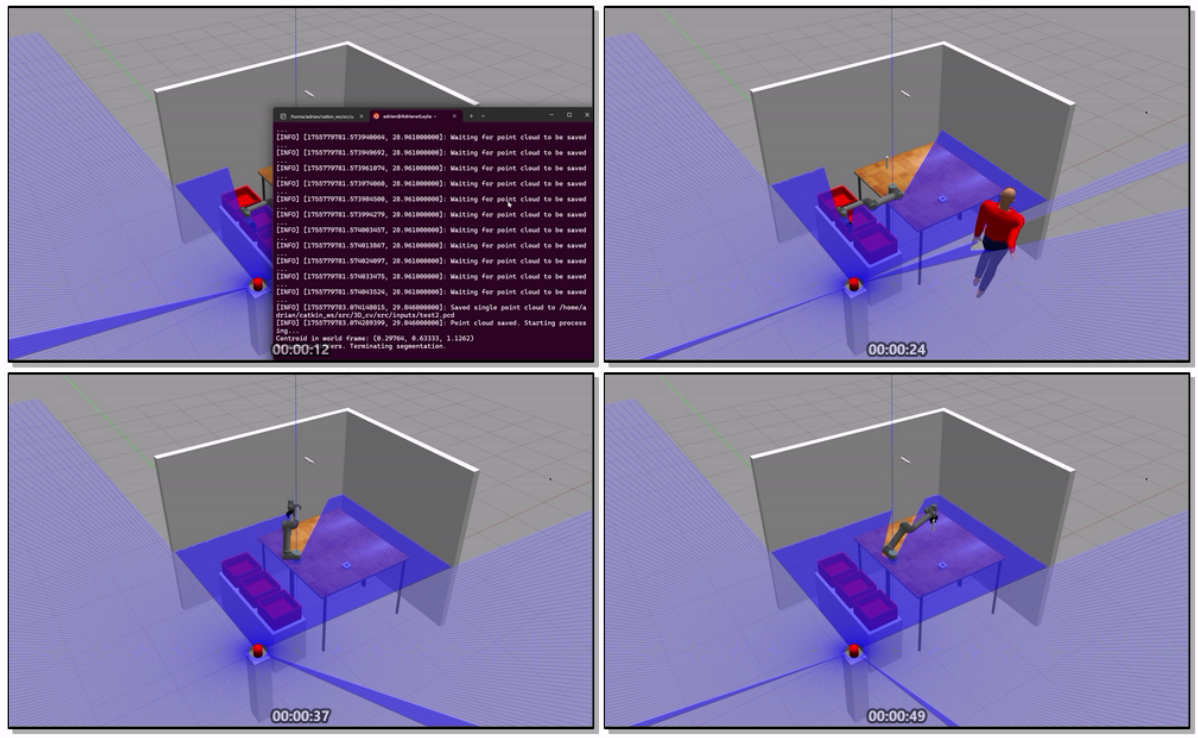

Figure 1: Sequential snapshots extracted from the task video, illustrating key stages of the pick-and-place operation during the simulated assembly process. These frames highlight the robot's motion trajectory, object interaction, and alignment behavior throughout Task 1

For a complete visualization of the task execution, including full motion sequences and pick-and-place operations, the video can be accessed via the following Google Drive link:

View full task video on Google Drive

The video includes both task 1 and task 2, demonstrating the interaction of the robot with cylindrical components and validating the safety procedures of the system under collaborative

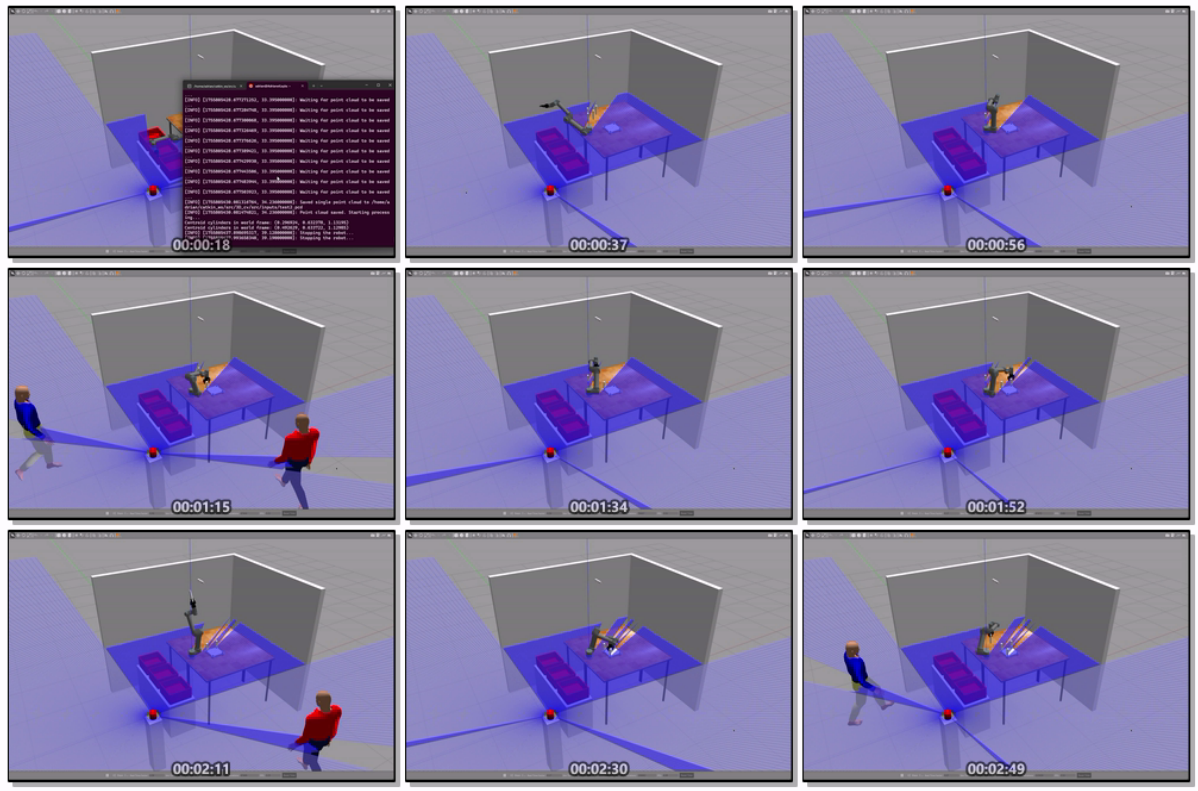

Figure 2: Sequential snapshots extracted from the task video, illustrating key stages of the pick-and-place operation during the simulated assembly process. These frames highlight the robot's motion trajectory, object interaction, and alignment behavior throughout Task 2

human-robot conditions.