Chair of Computational Modeling and Simulation
TUM School of Engineering and Design
Technical University of Munich

Framework for Robotic Fabrication and Automation in Timber Construction using COMPAS FAB

Scientific work to obtain the degree

Master of Science (M.Sc.)

at the TUM School of Engineering and Design
of the Technical University of Munich.

Supervised by Mohammad Reza Kolani
Dr. Stavros Nousias

Submitted by Alicia Knauer
Fraunhoferstr. 15
D-80469 München
e-Mail: alicia.knauer@tum.de

Submitted on 17. February 2025

Acknowledgment

I would like to express my gratitude to my supervisors, Mohammad Reza Kolani and Dr. Stavros Nousias, for their invaluable guidance, support, and encouragement throughout this research journey. Their insights and expertise have been instrumental in shaping this thesis. I also extend my appreciation to my professors and colleagues at TUM for their constructive feedback and thought-provoking discussions. A heartfelt thank you to my partner, my family and friends for their belief in me, offering support and motivation.

Abstract

The adoption of robotic fabrication and automation in timber construction offers significant potential for improving efficiency, precision, and sustainability in the construction industry. This thesis investigates the role of simulation in advancing automated construction workflows, utilizing COMPAS FAB as a computational tool for parametric design and robotic path planning. The study examines key challenges in automation, such as labor shortages, safety concerns, and material waste, and explores how robotic processes can address these issues through enhanced precision and adaptability. Through computational modeling and robotic simulations, this research evaluates the feasibility of automated timber assembly. The findings highlight the benefits of robotic fabrication in optimizing material use, improving accuracy, and increasing scalability within timber construction. This work contributes to the ongoing development of digital fabrication techniques and reinforces the importance of simulation in advancing automation in architecture and manufacturing.

Zusammenfassung

Die Einführung von Roboterfertigung und Automatisierung im Holzbau bietet ein erhebliches Potenzial zur Verbesserung von Effizienz, Präzision und Nachhaltigkeit in der Bauindustrie. In dieser Arbeit wird die Rolle der digitalen Simulation bei der Förderung automatisierter Bauabläufe untersucht, wobei COMPAS FAB als Berechnungswerkzeug für parametrisches Design und robotergestützte Bahnplanung eingesetzt wird. Die Studie untersucht die wichtigsten Herausforderungen bei der Automatisierung, wie z. B. Arbeitskräftemangel, Sicherheitsbedenken und Materialverschwendung, und untersucht, wie Roboterprozesse diese Probleme durch verbesserte Präzision und Anpassungsfähigkeit lösen können. Anhand von Computermodellen und Robotersimulationen wird in dieser Studie die Machbarkeit der automatisierten Holzmontage untersucht. Die Ergebnisse zeigen die Vorteile der Roboterfertigung bei der Optimierung des Materialeinsatzes, der Verbesserung der Genauigkeit und der Erhöhung der Skalierbarkeit im Holzbau. Diese Arbeit trägt zur laufenden Entwicklung digitaler Fertigungstechniken bei und unterstreicht die Bedeutung der Simulation bei der Förderung der Automatisierung in Architektur und Fertigung.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The construction industry is a critical sector of the global economy, employing approximately 7% of the workforce and generating annual expenditures of nearly 10 trillion Dollars. Despite its vast scale, the industry faces persistent challenges, including low productivity, labor shortages, and high accident rates, with construction-related fatalities accounting for 21% of workplace deaths (CAI and ZOU, 2022). Traditional construction methods often rely on intensive manual labor, leading to inefficiencies and safety risks. In response, the adoption of robotic fabrication and automation is emerging as a transformative approach to improve precision, reduce physical strain on workers, and enhance overall efficiency.

Timber construction, in particular, stands to benefit from robotic automation due to its modular nature and sustainability advantages. By integrating robotic fabrication with digital design tools, such as Building Information Modeling (BIM), the construction process can be streamlined, reducing material waste and enhancing structural accuracy (LIU et al., 2021).

Recent advancements in robotic technologies have spurred research into their applications in large-scale construction. However, challenges persist in simulating robotic workflows, integrating automation with existing industry practices, and developing standardized tools for seamless human-robot collaboration (XIAO et al., 2023). The COMPAS FAB framework addresses these gaps by providing a Python-based computational framework for robotic path planning and simulation (RUST et al., 2018).

Additionally, robotic fabrication offers the potential to significantly improve modular construction methods by enabling precise prefabrication, reducing human error, and ensuring greater consistency across structural components. Modular construction, which has gained popularity due to its efficiency and sustainability benefits, aligns well with robotic assembly processes. By pre-assembling timber components in controlled environments, waste is minimized, and quality control is improved (ZADEH et al., 2018). Furthermore, integrating robotic systems with parametric design tools allows for more adaptable and customized architectural solutions, facilitating innovative structural forms that would be difficult to achieve using traditional methods (ALFIERI et al., 2020).

Despite the promise of robotic fabrication in timber construction, several barriers remain. The initial investment in robotic systems, including hardware, software, and training, poses a significant challenge for widespread adoption. Moreover, there is a need for enhanced interoperability between robotic simulation platforms, BIM models, and physical

manufacturing workflows (SAIDI et al., 2016). Addressing these challenges will be crucial in realizing the full potential of robotics in timber construction and ensuring the scalability of automated fabrication techniques.

This thesis investigates the potential of COMPAS FAB in automating timber construction, offering insights into its capabilities, limitations, and future prospects. By analyzing case studies, conducting experimental simulations, and exploring industry applications, the research aims to provide a comprehensive understanding of how robotics can revolutionize the timber construction industry.

1.2 Outline

This thesis explores the integration of robotic fabrication and automation in timber construction using the COMPAS FAB framework. It is structured to provide a comprehensive understanding of the subject through several key chapters. Chapter 2 provides an in-depth review of the existing literature and theoretical foundations relevant to robotic fabrication and automation in construction. It begins with an overview of construction automation and robotics (CAR), discussing technological advancements in prefabrication, on-site robotics, 3D printing, and AI-driven systems. The benefits of CAR, including enhanced efficiency, sustainability impacts, and safety improvements, are highlighted.

The chapter then delves into the role of collaborative robots (cobots) in construction, emphasizing their differences from traditional industrial robots and their benefits in terms of efficiency, precision, flexibility, and reduced physical strain on workers. The integration of cobots with augmented reality (AR) interfaces, mobile robotic systems, and cooperative assembly techniques is discussed, with examples from recent advancements and research.

Next, the chapter covers BIM-based approaches for automation in construction, highlighting the advantages of off-site construction (OSC) in terms of productivity, quality, and waste reduction. The challenges in the design process for building cladding in OSC are addressed, and a BIM-based generative framework is introduced, including a building information extraction module, a generative design algorithm, and a simulation-based performance evaluation model, with a case study by LIU et al., 2021.

The chapter concludes with a discussion on timber as a sustainable construction material, covering its structural behavior, durability, environmental benefits, and the role of modular construction in enhancing design and drawing processes. The challenges and limitations of robotic fabrication in construction are also examined.

Chapter 3 outlines the research methodology employed in the study. It begins with an overview of the method, describing the integration of parametric design tools with robotic control systems to enable precise planning and execution of timber assemblies. The software and tools used in the research, including Rhino, Grasshopper, ROS, MoveIt!, and COMPAS FAB, are detailed. The chapter also covers the fundamentals of robotics

necessary for motion planning, such as coordinate frames, transformations, forward and inverse kinematics, robotic manipulators, kinematic models, motion planning, planning scenes, collision detection, and the Unified Robot Description Format (URDF).

Chapter 4 describes the practical implementation of the research. It begins with the design intent and functional requirements of the modular timber joinery system, followed by the design and parameters of the timber components and the manufacturing process. The chapter then covers the setup and customization of a robotic simulation framework using COMPAS FAB in Grasshopper. This includes the creation of custom URDF files for the UR10 robot model with the Robotiq Hand-E adaptive gripper, generating a MoveIt! configuration package, and the creation of a new Docker image with the custom configuration. The conversion of structural data from STEP to JSON format for import into Grasshopper is also detailed. The chapter concludes with an overview of the Grasshopper Playground as an interactive control interface for robotic simulation, describing the robot workflow for a single pick-and-place cycle and the Grasshopper component blocks used for simulated robotic assembly.

Chapter 5 presents the experimental evaluation of the robotic fabrication process. It includes a detailed analysis of the results obtained from the implementation phase, highlighting the effectiveness and efficiency of the proposed methods. The chapter discusses the performance of the robotic system in executing the assembly tasks, evaluating the overall feasibility of the approach, and identifying areas for improvement.

Chapter 6 summarizes the key findings of the research and provides conclusions based on the experimental results. It outlines the contributions of the study to the field of robotic fabrication and automation in timber construction and suggests potential areas for future research and development. The chapter discusses possible improvements and extensions to the current work, emphasizing the importance of continued innovation and exploration in this area.

Chapter 2

State of the Art

2.1 Robotic Fabrication and Automation in Construction

The integration of robotic fabrication and automation is transforming construction by enhancing precision, reducing material waste, and addressing labor shortages (PAN et al., 2018; SAIDI et al., 2016). As construction projects grow in complexity, automation technologies, including prefabrication systems, on-site robotics, and AI-driven tools, are being increasingly adopted to improve efficiency and sustainability (EVERSMANN et al., 2017).

This section examines key developments in this field. Construction Automation and Robotics discusses advancements such as prefabrication, robotic on-site factories, large-scale 3D printing, and AI-driven project management, along with their implications for sustainability and industry adoption (PAN et al., 2018; SAIDI et al., 2016; TANNE and INDRAYANI, 2023).

By analyzing these advancements, this section provides a framework for understanding how robotic fabrication and automation contribute to the transformation of construction methodologies while addressing challenges related to cost, regulation, and workforce adaptation (PAN et al., 2018; SAIDI et al., 2016).

2.2 Construction Automation and Robotics

The construction industry is undergoing a paradigm shift with the integration of automation and robotics, leading to enhanced efficiency, sustainability, and safety. Construction Automation and Robotics (CAR) encompasses various technologies, including prefabrication, on-site robotics, 3D printing, and artificial intelligence (AI)-driven systems. These innovations are transforming traditional construction methods, reducing labor dependency, and improving project sustainability. CAR examines technological advancements, sustainability impacts, and adoption challenges, drawing from the frameworks of PAN et al., 2018, SAIDI et al., 2016, TANNE and INDRAYANI, 2023, and EVERSMANN et al., 2017.

The evolution of CAR from mechanization to AI-driven autonomous systems has led to significant breakthroughs. The use of robotics in prefabrication facilitates mass customization and reduces construction waste (PAN et al., 2018), while automated production lines in modular construction enhance productivity and quality control (SAIDI et al., 2016). On-site robotics, such as single-task robots performing repetitive tasks like bricklaying and excavation, minimize human intervention (PAN et al., 2018), and autonomous robotic on-

site factories (AROFs) streamline construction through integrated robotic systems (SAIDI et al., 2016). Additionally, large-scale 3D printing accelerates housing and infrastructure development by reducing material costs and environmental impact (PAN et al., 2018), and robotic concrete printing enhances precision and structural integrity (SAIDI et al., 2016). The integration of AI-driven project management tools further optimizes scheduling, risk assessment, and resource allocation (PAN et al., 2018), while digital twin technology enables real-time monitoring and predictive analytics for construction sites (SAIDI et al., 2016). Furthermore, robotic prefabrication methods are advancing in timber structures, promoting efficient and precise large-scale spatial assembly (EVERSMANN et al., 2017).

The sustainability assessment framework (CARSAM) proposed by PAN et al., 2018 evaluates CAR's contributions across environmental, economic, and social dimensions. Automation reduces material waste through precision engineering (PAN et al., 2018), and AI-driven energy optimization lowers carbon emissions and enhances resource efficiency (SAIDI et al., 2016). Increased automation leads to lower labor costs and improved project timelines (PAN et al., 2018), and CAR enhances return on investment by reducing construction delays (SAIDI et al., 2016). Robotics improve worker safety by minimizing exposure to hazardous environments (PAN et al., 2018), while high-tech job creation offsets concerns over traditional labor displacement (SAIDI et al., 2016). Additionally, an assessment of automation in Indonesian state-owned construction enterprises highlights gaps in best practices and potential applications in different project life cycle stages (TANNE and INDRAYANI, 2023).

Despite its benefits, CAR faces several significant challenges. The adoption of robotic construction systems requires substantial capital investment, which may deter small and medium enterprises (PAN et al., 2018). Specialized training and expertise are necessary for operating and maintaining CAR technologies (SAIDI et al., 2016), while the absence of standardized regulations for construction robotics hinders large-scale implementation (PAN et al., 2018). Furthermore, the construction sector's reliance on traditional methods slows technological adoption (SAIDI et al., 2016).

The future of CAR lies in increased AI integration, cost-effective solutions, and regulatory advancements. Emerging trends such as collaborative robots (cobots), AI-driven automation, and smart building technologies are expected to drive industry transformation. Addressing financial and regulatory challenges will be crucial to ensuring widespread CAR adoption and maximizing its potential benefits. By aligning with sustainability goals and improving efficiency, CAR represents a fundamental shift towards a more resilient and productive construction sector.

2.2.1 Collaborative Robots in Construction

Collaborative robots (cobots) are increasingly transforming the construction industry by enabling human-robot cooperation in complex fabrication and assembly tasks. Unlike traditional industrial robots, which are typically confined to controlled environments and require strict safety measures, cobots are designed to interact with human workers dynamically.

They enhance efficiency, precision, and flexibility while reducing physical strain on human workers and minimizing errors in construction processes. Recent advancements focus on the integration of cobots with augmented reality interfaces, mobile robotic systems, and cooperative assembly workflows, demonstrating the potential of robotic assistance in various construction applications (ALEXI et al., 2024).

A significant development in collaborative robotics is the integration of augmented reality (AR) interfaces to facilitate more intuitive interactions between humans and robots. AMTSBERG et al., 2021 present the interactive human-robot collaboration (iHRC) system, which leverages AR for real-time coordination and task-sharing in construction settings. This system allows human workers to visualize robotic tasks in advance, make adjustments as needed, and interact seamlessly with robots, thereby reducing errors and improving workflow efficiency. Similarly, KYJANEK et al., 2019 demonstrate how AR-assisted robotic prefabrication in timber construction enables human workers to monitor, guide, and adjust robotic actions during fabrication. These AR-enhanced collaborations make construction processes more adaptable and precise, ensuring that human expertise is fully utilized while robots handle repetitive and physically demanding tasks.

Another critical advancement in collaborative robotics is the development of mobile robotic systems that assist with construction tasks in real-world, unstructured environments. DÖRFLER et al., 2016 introduce a mobile robotic bricklaying system that autonomously places bricks while working alongside human operators. The system combines robotic precision in material placement with human expertise in adapting to site-specific challenges, leading to increased efficiency and reduced material waste. Similarly, interactive robotic plastering, as explored by MITTERBERGER, ERCAN JENNY, et al., 2022, utilizes mobile robots to assist in on-site plaster application. This system integrates real-time feedback and human input, allowing for adaptive surface finishing and improved material consistency. By leveraging mobile robotic solutions, construction projects can reduce manual labor intensity, minimize inconsistencies, and improve overall project timelines.

Beyond AR interfaces and mobile robots, researchers are exploring cooperative assembly techniques, where humans and robots work together to construct intricate architectural elements. MITTERBERGER, ATANASOVA, et al., 2022 investigate human-robot collaboration in assembling wooden structures using rope joints, demonstrating how robotic precision can enhance structural integrity while allowing human workers to focus on complex decision-making and creative adjustments. The research highlights the benefits of robotic assistance in handling flexible materials, where human dexterity is complemented by robotic accuracy.

These cooperative workflows extend beyond traditional construction methods, allowing for new design possibilities that would be challenging to achieve with human labor alone. For instance, robots can manipulate materials in ways that improve structural performance, while humans oversee and guide the assembly process. This symbiosis fosters an innovative approach to architectural design and fabrication, where robotic capabilities are fully integrated into construction methodologies.

As collaborative robotic systems become more advanced, their role in construction is expected to grow, particularly in areas such as automated assembly, adaptive fabrication, and site automation. The integration of real-time data analysis, machine learning, and digital design tools will further enhance human-robot collaboration, allowing for more efficient, safe, and scalable construction processes. Future developments may include robotic systems that learn from human workers, automated quality control through AI-driven inspections, and greater autonomy in material handling and assembly.

Overall, the research by AMTSBERG et al., 2021, DÖRFLER et al., 2016, KYJANEK et al., 2019, and MITTERBERGER, ATANASOVA, et al., 2022 highlights the potential of collaborative robots in shaping the future of construction. These systems are not merely tools for automation but rather active partners in the construction process, enabling new forms of craftsmanship, efficiency, and safety. By integrating AR, mobile robotics, and cooperative workflows, the construction industry is moving toward a future where humans and robots seamlessly collaborate to achieve higher levels of precision and innovation.

2.2.2 BIM-based Approaches for Automation in Construction

The integration of Building Information Modeling (BIM) with Design for Manufacturing and Assembly (DfMA) principles has gained significant attention as a means of enhancing automation in construction. This approach is particularly relevant in the context of panelized building design, where the automation of both the design and manufacturing processes can result in increased efficiency and precision. Liu et al. LIU et al., 2021 propose a BIM-enabled generative framework for building panelization design, which utilizes BIM data to automate the design of production components. This framework not only facilitates more accurate design but also optimizes the selection and organization of materials for prefabrication, thereby reducing errors and inefficiencies in the manufacturing process.

ALFIERI et al., 2020 investigate the integration of BIM and DfMA within an off-site construction framework, focusing on the Italian context. Their study demonstrates the potential of BIM to support automated workflows for panelized construction by providing a structured approach to design and manufacturing processes. BIM's role in improving the coordination between design and manufacturing is particularly crucial in DfMA-based systems, where precision and minimal material waste are essential.

YUAN et al., 2018 further explore the role of parametric design in the automation of prefabricated buildings, highlighting the benefits of a DfMA-oriented parametric approach. By using parametric modeling, this study demonstrates how automation can address fabrication constraints while enhancing design flexibility. The parametric design tools are capable of generating efficient designs that consider both structural requirements and manufacturing constraints, aligning closely with the generative framework proposed by LIU et al., 2021.

The work of ALWISY et al., 2019 also contributes to the discussion by focusing on BIM-based automation for the design and drafting of wood panels for modular residential

buildings. Their research underscores the utility of BIM in automating the creation of detailed manufacturing designs, which helps to reduce manual drafting errors and improve the overall speed of production. This BIM-driven approach aligns with the growing emphasis on improving design-to-manufacturing workflows through automation.

In the context of mass timber construction, ZADEH et al., 2018 explore the integration of BIM with DfMA principles, highlighting how BIM can be used to optimize the design and fabrication of timber components. The authors discuss how BIM's ability to manage complex geometries and material specifications facilitates the automation of timber panelization, ensuring both accuracy and efficiency in the construction process.

In addition, ZADEH et al., 2018 identify both the challenges and opportunities of applying BIM-based design and fabrication methods to timber construction. Their work emphasizes the need for integrated design and manufacturing systems that enhance efficiency and address the unique challenges of timber as a building material.

In conclusion, the integration of BIM with DfMA and parametric design principles plays a pivotal role in advancing automation in construction, particularly in the context of panelized building systems. By automating design, manufacturing, and assembly processes, BIM-based approaches contribute to the reduction of material waste, improved design precision, and enhanced efficiency in off-site construction processes (LIU et al., 2021; ALFIERI et al., 2020; YUAN et al., 2018; ALWISY et al., 2019; ZADEH et al., 2018).

2.3 Timber as a Sustainable Construction Material

2.3.1 Physical and Mechanical Properties of Timber

Timber is widely used in construction due to its favorable physical and mechanical properties. One of its most significant advantages is its high strength-to-weight ratio, which makes it much lighter than concrete and steel while still providing substantial load-bearing capacity (BREYER et al., 2019). This characteristic allows for easier handling, transportation, and installation, which is particularly beneficial for prefabricated and modular construction methods (KEEPING and SHIERS, 2017).

Another key property of timber is its elasticity and structural flexibility, allowing it to absorb stresses from wind loads, earthquakes, and other environmental factors. Moreover, the natural thermal and acoustic insulation properties of timber contribute to improved indoor comfort. Its low thermal conductivity reduces energy demands for heating and cooling, while its porous structure effectively absorbs sound, making it ideal for residential and commercial buildings (HERZOG et al., 2012).

Despite these advantages, timber is susceptible to environmental factors such as moisture, biological decay, and insect damage. The development of engineered wood products, such as cross-laminated timber (CLT) and glue-laminated timber (Glulam), further improves

timber's structural performance, reducing issues related to natural defects like knots and warping (KEEPING and SHIERS, 2017).

2.3.2 Timber Construction Methods

Timber construction methods have evolved significantly, ranging from traditional framing techniques to modern prefabricated systems. Traditional timber framing, which relies on interlocking beams and joints, has been used for centuries and is still valued for its aesthetic appeal and durability (BREYER et al., 2019). Post-and-beam construction, often seen in historical buildings, employs heavy timber elements to form a stable structure without the need for metal fasteners.

Platform framing, where each floor is constructed separately and stacked on the one below, provides a highly efficient and modular approach. Balloon framing, an older technique, features long vertical studs running continuously from foundation to roof, though it is less commonly used today due to challenges in material waste.

Advancements in prefabricated and modular timber construction have revolutionized the industry by improving precision and reducing on-site labor costs. Off-site manufacturing allows for higher quality control, minimized material waste, and faster assembly times (KEEPING and SHIERS, 2017; HERZOG et al., 2012). Mass timber construction, which utilizes large prefabricated wood elements, is becoming increasingly popular due to its strength, stability, and speed of installation.

Half-Timbered Houses

Half-timbered houses represent one of the oldest and most recognizable timber construction methods, particularly prevalent in medieval European architecture. These structures consist of a wooden framework with spaces infilled with materials such as wattle and daub, brick, or plaster (CAMPBELL, 2019; HARRIS, 1993). The exposed timber elements form a distinct aesthetic while providing essential structural support.

A key advantage of half-timbered construction is its flexibility and ease of repair. The modular nature of the framework allows for individual timber beams to be replaced without dismantling the entire structure (HARRIS, 1993). Additionally, the wooden framework distributes loads efficiently, making these buildings highly durable despite their historical origins. The use of diagonal bracing further enhances stability, particularly in regions prone to high winds or seismic activity (CAMPBELL, 2019).

While half-timbered houses are not as common in contemporary construction, they remain significant in historical preservation and restoration projects. Advances in timber treatments and protective coatings have enabled modern adaptations of this technique, blending traditional craftsmanship with improved durability and fire resistance (HARRIS, 1993).

Timber remains an essential material in construction due to its favorable strength-to-weight ratio, thermal performance, and adaptability to various construction methods. Traditional timber framing techniques persist in certain applications, while modern platform framing, prefabrication, and mass timber construction are driving innovation in the industry. Half-timbered houses continue to hold cultural and architectural significance, offering insights into historical construction methods. As research and technology continue to evolve, timber's role in construction is expected to expand, offering a balance of efficiency, performance, and design versatility (BREYER et al., 2019; KEEPING and SHIERS, 2017).

2.3.3 Mortise and Tenon Joints in Timber Construction

Mortise and tenon joints have long been a critical component in timber construction due to their mechanical strength and reliability in joining structural elements. This joint consists of a mortise, a cavity or hole, and a tenon, a protruding section that fits into the mortise, creating a robust connection between two pieces of timber. The application of mortise and tenon joints spans various fields, including furniture making, traditional timber framing, and modular systems. Common variations of this joint include through tenons, which extend through the full thickness of the timber; blind mortises, which do not penetrate the timber entirely; and pegged tenons, which incorporate dowels or pegs for additional reinforcement (HASSAN et al., 2023;FEIO et al., 2014).

Historically, mortise and tenon joints were crafted manually, relying on skilled labor and tools. However, technological advancements, particularly the use of CNC routers and milling machines, have transformed this process. These modern techniques allow for greater precision, repeatability, and efficiency in the production of joints, particularly in prefabricated modular systems where uniformity is essential. CNC milling, for example, facilitates the consistent creation of complex joints, reducing errors and material waste (SCHMIDT and DANIELS, 1999). Furthermore, specialized drilling techniques, such as hollow chisel mortising and CNC mortisers, enhance the accuracy of square mortises, an essential feature for achieving strong, well-fitted joints in timber construction (FEIO et al., 2014). Mortises can also be manufactured with mortisers, which are specifically designed tools for cutting precise square or rectangular holes, making them highly effective for creating mortises in timber joinery (FEIO et al., 2014). These advancements in machine-assisted joinery complement traditional craftsmanship, maintaining the structural integrity and aesthetic qualities of mortise and tenon joints in both contemporary and historic timber applications.

CNC-milled joinery systems have significantly advanced woodworking by automating the cutting of joinery with enhanced precision and efficiency. CNC routers and milling machines are programmed to perform complex cutting tasks, ensuring high accuracy and reducing human error. This automation allows for increased customization, enabling tailored designs for specific applications, such as modular construction and custom furniture. Additionally, the repeatability of CNC systems ensures consistent quality across multiple units, making them ideal for mass production and large-scale projects. As a result, CNC technology

has become essential in modern woodworking, offering both precision and efficiency (QUESADA, 2005; OVERBY, 2010).

Understanding Mortise and Tenon joints is essential, as this technique will be pivotal in the custom-designed timber components discussed in the implementation chapter.

2.3.4 Environmental Benefits of Timber

Timber is increasingly regarded as a sustainable construction material due to its renewable nature and potential to reduce the environmental impact of buildings. Its use in construction has garnered considerable attention, driven by growing concerns about climate change and the need for more sustainable building materials. Compared to conventional materials like concrete and steel, timber offers several environmental advantages, such as carbon sequestration, lower embodied energy, and potential for reuse.

One of the key environmental benefits of timber is its ability to sequester carbon dioxide. Trees absorb carbon dioxide from the atmosphere during photosynthesis, storing it in their biomass. When used in construction, timber effectively locks away this carbon for the duration of the building's life. This characteristic significantly reduces the overall carbon footprint of buildings constructed with timber. Studies have shown that timber buildings have a reduced net carbon footprint compared to conventional alternatives, such as those built with steel and concrete, as they not only store carbon but also require less energy-intensive processing (CHEN et al., 2020). Thus, timber acts as a long-term carbon sink, contributing to the mitigation of climate change by decreasing atmospheric carbon dioxide levels.

Life cycle assessment (LCA) is an essential tool for evaluating the environmental impacts of construction materials over their entire life cycle. Several studies comparing the LCA of timber to more conventional building materials have revealed that timber generally has a lower environmental impact. Specifically, timber is more energy-efficient during its production phase and is renewable, which reduces its overall environmental footprint. Hart and Pomponi HART and POMPONI, 2020 highlight that timber buildings tend to have lower embodied energy and a reduced global warming potential compared to concrete and steel structures. These benefits arise from timber's renewability and lower processing energy requirements. In particular, timber structures, such as cross-laminated timber (CLT) and glue-laminated timber (glulam), perform better across most environmental impact categories in LCA studies, making them an attractive alternative in sustainable building design.

Modern timber construction methods, such as cross-laminated timber (CLT) and glue-laminated timber (glulam), represent significant innovations that enhance timber's performance as a construction material. CLT, for example, is manufactured by layering wooden planks in alternating directions, creating a strong, durable material suitable for multi-story buildings. This innovation allows timber to be used in a broader range of construction projects, offering a sustainable alternative to concrete and steel YOUNIS and DODOO, 2022.

Similarly, glulam is a form of engineered wood that provides high strength and versatility, enabling the creation of large-scale structures with minimal material use. D'Amico et al. D'AMICO et al., 2021 emphasize that mass timber systems, such as CLT and glulam, can replace higher-carbon materials, such as reinforced concrete, resulting in significant reductions in embodied carbon. These innovations make timber not only a sustainable choice but also a competitive one for modern construction projects.

Timber is inherently well-suited to the principles of the circular economy due to its renewability and ability to be reused and recycled. Unlike concrete or steel, which require significant energy to recycle and have limited reuse potential, timber can be repurposed multiple times, extending its life cycle. Padilla-Rivera et al. PADILLA-RIVERA et al., 2018 explore how wood-frame construction, through its capacity for carbon reduction and waste minimization, contributes to a more circular approach in the built environment. Timber can be reused in various forms, such as in the construction of new buildings, or recycled into products like paper or composite materials. This reduces the overall demand for raw materials and minimizes waste, aligning with the goals of sustainable and circular building practices.

Despite its benefits, timber does face several challenges in construction, including issues related to durability, fire resistance, and sustainable sourcing. These challenges can limit its widespread adoption in some building applications. However, recent advancements in timber treatment and construction techniques have made it possible to address these concerns. For instance, modern fire-retardant treatments can significantly improve the fire resistance of timber structures, making them suitable for use in more fire-sensitive contexts. Additionally, ensuring that timber is sourced sustainably is a critical consideration. Certification schemes, such as the Forest Stewardship Council (FSC), help guarantee that timber is harvested responsibly, ensuring that forest ecosystems are maintained HART and POMPONI, 2020. Ongoing research into improving the durability and fire safety of timber, alongside sustainable sourcing practices, is essential for mitigating these challenges.

Looking forward, the role of timber in construction is expected to expand as the demand for sustainable building materials grows. Timber's carbon sequestration potential, combined with innovations in construction techniques, positions it as a key material in the transition toward eco-friendly and energy-efficient buildings. As building codes and regulations become increasingly focused on sustainability, the use of timber is likely to increase, particularly in mid- to high-rise buildings where mass timber systems such as CLT and glulam provide an alternative to concrete and steel HART and POMPONI, 2020. The continued development of timber-based solutions, coupled with technological advancements in timber treatment, will enable timber to play a pivotal role in shaping the future of the construction industry.

Timber offers several significant environmental benefits, including carbon sequestration, reduced life cycle environmental impacts, and a key role in the circular economy. With advancements in innovative construction techniques, such as CLT and glulam, timber is becoming an increasingly viable option for sustainable construction. While challenges

related to durability, fire resistance, and sourcing persist, ongoing research and technological development are making these issues more manageable. As the construction industry continues to prioritize sustainability, timber's role as an eco-friendly material is expected to grow, making it a cornerstone of future low-carbon buildings.

2.4 Modular Construction

Modular construction has garnered considerable attention in recent years due to its potential to enhance efficiency, reduce costs, and improve sustainability in the building sector. This method involves the prefabrication of building components in a controlled factory setting, followed by on-site assembly. This approach contrasts with traditional construction techniques and offers unique benefits and challenges. Numerous studies have focused on the design, sustainability, and life cycle performance of modular construction, with an emphasis on integrating advanced technologies such as Building Information Modeling (BIM) to optimize these processes.

One of the key advantages of modular construction is its ability to streamline the design and manufacturing processes. The use of modular components allows for greater standardization, enabling faster construction times and reducing material wastage. SMITH, 2011 notes that modular construction's adaptability to diverse design requirements, while maintaining standardization, significantly enhances efficiency in building projects. Furthermore, technological advances, such as Building Information Modeling (BIM), have further optimized the design and drafting processes for modular buildings. ALWISY et al., 2019 explore the role of BIM in automating the design and drafting of wood panels for modular residential buildings. Their research demonstrates that BIM can improve design accuracy, reduce manual errors, and enhance the efficiency of manufacturing processes, making modular construction more scalable and precise. BIM also facilitates the customization of modular units, enabling tailored solutions to meet specific project requirements while still benefiting from standardization (ALWISY et al., 2019).

The environmental advantages of modular construction are significant. KAMALI and HEWAGE, 2015 emphasize that modular buildings typically have a lower environmental impact compared to traditional buildings, particularly in terms of energy use, material waste, and carbon emissions. The controlled factory setting in which modular components are produced ensures minimal material wastage, precise manufacturing, and optimized energy use. Additionally, modular buildings are well-suited to incorporate renewable energy systems, further enhancing their sustainability. LAWSON and OGDEN, 2010 underscore that the prefabrication process, when combined with energy-efficient materials and processes, can substantially reduce carbon emissions throughout the construction and operational phases of a building.

The long-term performance of modular buildings, including maintenance and energy consumption, is another critical aspect of their sustainability. KAMALI and HEWAGE, 2015 provide a detailed comparison of the life cycle performance of modular versus

conventional buildings, showing that while modular buildings may incur higher initial costs due to the advanced manufacturing process, they tend to offer lower long-term costs, particularly regarding energy consumption, maintenance, and operational expenses. Modular buildings are often better insulated and more energy-efficient, which helps reduce ongoing operational costs. Furthermore, modular construction allows for easier adaptation and renovation, thus minimizing the need for complete demolition and supporting the principles of a circular economy.

In addition, LAWSON and OGDEN, 2010 suggest that the modular construction method's ability to facilitate disassembly and reuse of components at the end of the building's life contributes to its long-term sustainability. The ease with which modular buildings can be modified or repurposed extends their functional life and minimizes waste. This flexibility, combined with lower operational costs, makes modular buildings an attractive option for developers and occupants who prioritize both financial and environmental considerations.

Despite the numerous advantages, modular construction does face certain challenges. One of the key issues is the limitation in design flexibility due to the reliance on standardized modules. While modular construction offers high efficiency, the constraints imposed by pre-fabricated components may not be suitable for projects that require highly customized solutions. SMITH, 2011 acknowledges that although modular construction is highly adaptable in many instances, it may not be ideal for projects demanding significant design variation.

Logistical challenges, such as the transportation of large modular units to construction sites, can also result in increased costs and delays if not properly managed. The integration of advanced technologies such as BIM can mitigate some of these challenges by optimizing the planning and execution of the construction process. ALWISY et al., 2019 emphasize that BIM not only aids in the design process but also contributes to optimizing the entire construction workflow, including transportation logistics, which reduces inefficiencies and helps manage the practical aspects of modular construction more effectively.

In conclusion, modular construction presents significant potential for enhancing the efficiency, sustainability, and overall life cycle performance of buildings. The integration of advanced technologies such as BIM has further optimized modular design and manufacturing processes, making them more adaptable and precise. Research has shown that modular buildings generally have a lower environmental impact—especially in terms of energy use, material waste, and carbon emissions—compared to traditional construction methods. Additionally, their long-term life cycle costs are often lower, particularly when considering energy efficiency and ease of modification. However, challenges related to design flexibility and logistics persist, and continued innovation is required to overcome these limitations. Future research should focus on overcoming these challenges and further improving the environmental and economic performance of modular construction.

2.5 Challenges and Limitations of Robotic Fabrication in Construction

Robotic fabrication in construction offers numerous benefits, including increased efficiency, precision, and sustainability. However, several challenges and limitations hinder its widespread adoption and optimal performance. These challenges span technological, economic, environmental, and social dimensions, as highlighted by key studies in the field.

One of the primary challenges in robotic fabrication is the integration of automation within traditional construction workflows. SAIDI et al., 2016 emphasize that while robotic systems enhance efficiency and safety, they require advanced technological infrastructure and seamless coordination between various automated systems. Additionally, EVERSMANN et al., 2017 highlight the complexity of large-scale spatial assembly, particularly in timber construction, where material variability and adaptability present significant obstacles to automation. LIU et al., 2021 further discuss the integration of BIM-enabled generative design frameworks to support automated fabrication, yet challenges remain in interoperability and data consistency between digital and physical construction processes.

The high initial investment costs for robotic systems and the required digital infrastructure pose significant economic barriers. PAN et al., 2018 discuss the financial implications of construction automation, noting that while long-term cost savings are achievable, the upfront capital expenditure and operational costs can be prohibitive, especially for small and medium-sized enterprises (SMEs). Furthermore, the economic viability of robotic fabrication depends on the scale of production and project-specific factors, which can limit its broader application.

While robotic fabrication has the potential to enhance sustainability, challenges persist in material waste management and energy consumption. ALFIERI et al., 2020 examine how BIM and Design for Manufacture and Assembly (DfMA) can optimize automated workflows, yet inefficiencies in material handling and recycling processes remain. CHEN et al., 2020 and HART and POMPONI, 2020 explore the environmental benefits of timber construction, noting that while automation can support sustainable practices, the embodied carbon of robotic systems and their energy demands must be carefully assessed. Additionally, YOUNIS and DODOO, 2022 argue that life-cycle assessment methodologies need to be further developed to accurately quantify the sustainability benefits of robotic fabrication.

The adoption of robotic fabrication requires a workforce skilled in robotics, programming, and digital construction technologies. PAN et al., 2018 highlight the social implications of automation, including the need for reskilling and potential job displacement concerns. D'AMICO et al., 2021 emphasize the necessity for education and training programs to bridge the skill gap, ensuring that construction professionals can effectively collaborate with robotic systems. The transition from manual labor to automation also raises concerns about industry-wide acceptance and adaptation to new workflows.

The lack of standardized protocols and regulatory frameworks for robotic construction is another major limitation. PADILLA-RIVERA et al., 2018 point out that while sustainable construction methods, including timber fabrication, are gaining traction, regulatory barriers often slow down the adoption of automated techniques. The industry requires standardized guidelines for integrating robotics within construction processes, ensuring compliance with safety, quality, and environmental regulations.

In conclusion, while robotic fabrication in construction holds great promise, overcoming these challenges requires a multi-faceted approach involving technological advancements, economic strategies, sustainability considerations, workforce development, and regulatory support. Addressing these limitations will pave the way for a more efficient, sustainable, and automated construction industry.

Chapter 3

Methodology

Robotic fabrication in timber construction offers enhanced precision, efficiency, and automation. This method utilizes the COMPAS FAB framework (RUST et al., 2018) to integrate parametric design with robotic control, enabling seamless digital-to-physical workflows. The process involves computational design for structural feasibility, path planning, and simulation within a CAD environment. Key setup components include Docker-based ROS environments, URDF modeling for robot compatibility, and Grasshopper for real-time control. By combining these elements, the approach establishes a robust pipeline for automated timber assembly, improving accuracy and adaptability in robotic fabrication.

3.1 Method Overview

The proposed method aims to implement robotic fabrication for timber structures by utilizing the COMPAS FAB framework, a versatile computational platform for robotic and digital fabrication workflows. The approach integrates parametric design tools with robotic control systems, enabling precise planning and execution of timber assemblies. Initially, the timber structure's geometry is designed computationally, ensuring structural integrity and fabrication feasibility. The workflow incorporates path planning and simulation with the COMPAS FAB (RUST et al., 2018) environment in a CAD-Software to define robotic tasks. The method prepares for later use with physical robots to execute the fabrication process, achieving a seamless transition from digital design to physical construction. This approach enhances efficiency, precision, and flexibility in timber structure fabrication.

A critical component of the method is the intricate setup process, which encompasses configuring Docker, creating the URDF (Unified Robot Description Format) model, and integrating the system with Grasshopper. The setup begins with the installation and configuration of Docker to manage containerized ROS environments, including the definition of a custom Docker Compose file to orchestrate the required services. The next step involves generating the URDF model, which accurately describes the robot's physical structure, kinematics, and dynamics, ensuring compatibility with the ROS ecosystem. Finally, the workflow incorporates the use of Grasshopper, a visual programming environment, to connect the digital design process with robotic control. This stage involves establishing communication between Grasshopper and ROS with the help of COMPAS FAB, ensuring that the robotic system can execute fabrication tasks as planned. The integration of these components is essential for achieving a robust and functional robotic fabrication pipeline for working with timber structures and COMPAS FAB.

The components are generated in a CAD-environment and incorporated as meshes for the robot to assemble.

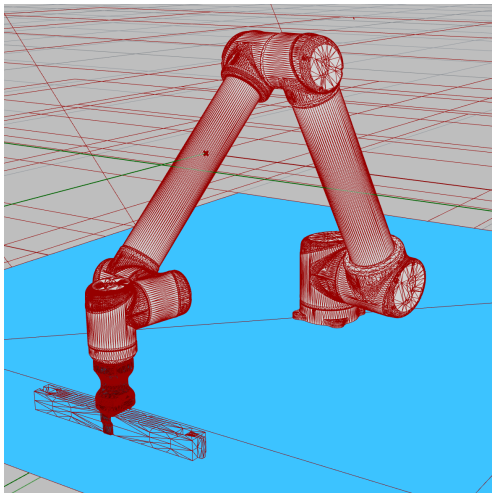


Figure 3.1: UR10 Simulation in Rhino

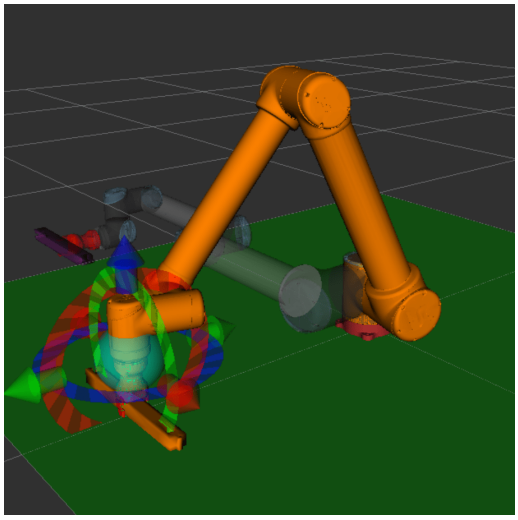


Figure 3.2: Visualization with Rviz

3.2 Software and Tools

3.2.1 Rhino and Grasshopper

Rhinoceros (Rhino) is a versatile computer-aided design (CAD) software widely utilized in various fields such as architecture, industrial design, and engineering. Renowned for its precise 3D modeling capabilities, Rhino supports the creation of complex geometries, allowing users to handle freeform surfaces, solid modeling, and polygon meshes with accuracy (McNeel, n.d.). Grasshopper, a visual programming environment that operates within Rhino, enhances this functionality by enabling parametric design. Through a node-based interface, Grasshopper allows users to manipulate design parameters algorithmically, offering a flexible and dynamic way to generate and modify geometries based on input variables. The Rhino and Grasshopper synergy is particularly beneficial in fields requiring custom geometries, iterative design processes, and optimization, as users can rapidly explore various design iterations and workflows, significantly improving efficiency (Tedeschi, 2014, McNeel, n.d., Woodbury, 2010). Within a Grasshopper Playground, an interactive environment within Grasshopper, users can explore "play" by connecting components without a specific end goal, often to test various design ideas or to familiarize themselves with how components work together. Grasshopper provides a wide range of pre-built components for mathematical operations, geometric transformations, data management, and input/output handling, which can be dragged onto the canvas and connected visually. As users change parameters or adjust connections, the output updates in real-time, allowing immediate feedback and exploration (R. McNeel, n.d.).

3.2.2 ROS, MoveIt! and RViz

ROS (ROS.ORG, n.d.-a), or the Robot Operating System, is an open-source framework that provides tools, libraries, and standards to streamline the creation of complex robot applications. As a middleware, it facilitates inter-process communication, allowing various independent components of a robot, such as perception, planning, and control, to interact seamlessly. This design enables easier management and integration of complex robotic systems by supporting distributed computing and communication through topics, services, and actions. ROS is widely used in both research and industry due to its robust ecosystem, which includes tools for simulation, visualization, testing, and debugging, all organized into modular packages that promote code sharing and reuse (QUIGLEY et al., 2009, ROS.ORG, n.d.-a).

MoveIt (PICKNIKINC., n.d.) is a ROS-based framework designed for motion planning, manipulation, 3D perception, kinematics, and control, making it particularly suited for robotic arms and manipulators. It streamlines the development of complex robotic behaviors by offering high-level interfaces for various motion-planning tasks, such as generating collision-free paths and optimizing trajectories. MoveIt also provides kinematic tools that calculate the joint positions needed for a robotic arm to reach specific positions in space. With integration for 3D perception, MoveIt can adjust a robot's path based on sensor input from cameras or LiDAR, and it supports both real-world control and simulation through ROS-compatible controllers (CHITTA et al., 2012, PICKNIKINC., n.d.).

RViz (ROS.ORG, n.d.-b) is a 3D visualization tool in ROS that enables real-time visualization of robot states and sensor data. It is particularly useful for debugging and monitoring robotic systems, as it allows developers to view data streams from various sources, such as LiDAR scans, camera images, and planned trajectories. RViz's interactive interface allows users to visualize robot models, adjust parameters like positions and orientations, and monitor changes in real time. This flexibility helps developers better understand and refine a robot's behavior within a complex environment. Additionally, RViz supports interactive markers, which provide a hands-on way to manipulate objects and test behaviors directly within the visualization (ROS.ORG, n.d.-b, COUSINS, 2012).

3.2.3 Universal Robots and Robotiq

Universal Robots (UR) is a company in the field of collaborative robotics, known for industrial robots that are versatile and safe for human-robot interaction. The company's robotic arms are designed to assist with a range of applications, including assembly, packaging, and quality inspection, across diverse industries such as manufacturing, healthcare, and electronics. Universal Robots' models—ranging from the lightweight UR3 to the larger UR16—are built for ease of programming, flexible deployment, and enhanced safety features, allowing for fast integration into various workflows (ROBOTS, n.d.).

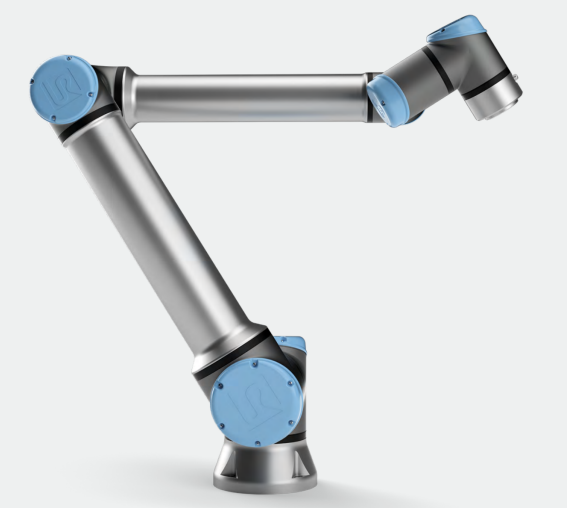


Figure 3.3: Universal Robot UR10e (UNIVERSAL ROBOTS, 2025)

The UR10, exemplifies these features with a payload capacity of 10 kg and a reach of 1300 mm, making it ideal for tasks requiring extended reach and moderate load capacity, such as palletizing and heavy-duty assembly. Its design emphasizes user-friendly operation and minimal setup time, encouraging efficient collaboration in environments traditionally dominated by manual labor (UNIVERSAL ROBOTS, 2025). These capabilities make the UR10 a suitable candidate for research in timber construction workflows by enabling detailed planning, reducing errors, and enhancing precision in assembling custom or complex timber elements.

Robotiq develops adaptive grippers that enable versatile robotic manipulation across various applications. Among its models, the Hand-E Adaptive Gripper is designed for precision tasks and adaptability in collaborative robotic systems. The gripper features a 50 mm parallel stroke, allowing it to handle a range of part sizes and shapes, with adjustable force settings for precision. Its design complies with ISO/TS 15066, incorporating safety measures such as rounded edges and self-locking mechanisms, which make it suitable for collaborative environments. Additionally, the Hand-E's sealed structure supports its operation in environments like CNC machining. These features make it a robust choice for research applications requiring precise and adaptive manipulation (ROBOTIQ, 2023).

3.2.4 COMPAS FAB - Python Package for Robotic Fabrication

COMPAS (MELE and many OTHERS, 2017-2021) is an open-source framework designed for computational design and digital fabrication, primarily targeting the architecture, engineering, and construction industries. It provides a rich set of tools for geometric computation, data manipulation, and visualization, empowering users to create complex geometries and design systems. The framework is built on Python and facilitates collaborative workflows through its modular architecture, allowing users to extend its capabilities with plugins tailored to specific applications (MELE and many OTHERS, 2017-2021).

COMPAS FAB is an extension of the COMPAS framework that focuses on the integration of robotic systems and digital fabrication processes. It offers a comprehensive set of tools for robotic motion planning, trajectory generation, and simulation, enabling users to develop automated workflows for various fabrication tasks, including milling, 3D printing, and assembly. By bridging the gap between computational design and robotic execution, COMPAS FAB enhances the potential for innovative fabrication strategies in architectural design and construction (RUST et al., 2018).

The implementation of this research was performed using the COMPAS version 1.17.10 and COMPAS FAB version 0.28.0.

Backends and ROS

A central feature of COMPAS FAB is its backend system, which includes integration with the Robot Operating System (ROS), a widely used middleware for robot control. Through its ROS backend, COMPAS FAB enables seamless interaction with ROS-compatible robots, facilitating real-time operations such as motion planning, collision detection, and robot control. This integration allows designers and engineers to bridge the gap between computational design and robotic execution, supporting automated workflows for advanced fabrication tasks such as milling, 3D printing, and assembly. By leveraging ROS's robust capabilities, COMPAS FAB empowers users to implement complex, collaborative, and adaptive robotic applications, streamlining the transition from design concepts to physical fabrication (RUST et al., 2018).

To make Integration and Set up easier, it is possible to leverage Docker for containerizing specific configurations. Docker is an open-source platform designed to simplify the development, deployment, and management of applications by using containerization. Containers are lightweight, standalone packages that include everything needed to run an application: the code, runtime, libraries, and dependencies. Unlike virtual machines, which require their own operating system, Docker containers share the host system's OS kernel, making them faster and more efficient (DOCKER, n.d.). ROS systems require multiple interconnected nodes, which can complicate deployment. Docker simplifies this by allowing users to create virtualized ROS networks, which manage the setup and connection of all nodes using a single configuration file. Running ROS nodes as containers ensures consistent performance across different systems, easing deployment and enhancing repeatability. Pre-built ROS images provide convenient starting points for these containerized setups (RUST et al., 2018). In this case, four different containers are used: ros-fileserver, ros-bridge, ros-core and moveit-demo which can all be found on Docker Hub.

Connecting COMPAS FAB with Grasshopper and Rhino

Through its robust API, COMPAS FAB enables seamless interoperability, allowing users to leverage Rhino and Grasshopper's geometric modeling capabilities within the COMPAS

ecosystem. This connection, as shown in figure 3.4, enables the user to develop and simulate robotic fabrication processes in a single integrated environment. In Rhino, COMPAS FAB extends traditional modeling capabilities by enabling real-time interactions with robotic systems, facilitating path planning and toolpath generation within a familiar CAD interface. Within Grasshopper, COMPAS FAB provides dedicated components that allow users to script, simulate, and test robotic tasks visually, making it accessible for both advanced users and those new to programming. The documentation provided by COMPAS FAB includes detailed guidance on connecting to various robotics systems, utilizing built-in kinematics solvers, and implementing modular workflows, thereby making it a versatile tool for robotic automation in architectural and industrial design applications (RUST et al., 2018). For this research purpose, a grasshopper playground is developed to control and modularize the robotic fabrication process for simple tasks in timber construction.

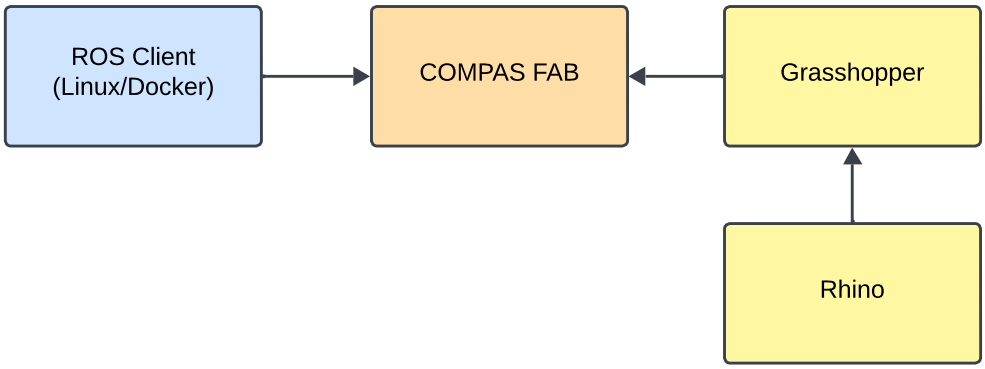


Figure 3.4: Relation between COMPAS FAB and software

3.3 Robotics Fundamentals for Motion Planning

3.3.1 Robotic Manipulators and Kinematic Models

A robotic manipulator consists of rigid links connected by joints — either revolute, which allow rotational motion, prismatic, which enable translational motion or fixed, which allow no motion. These joints and links form a kinematic chain, which can be open or closed. Open chains, commonly used in robotic manipulators, feature a sequential connection of links and joints that provide the degrees of freedom (DOF) required for mobility and flexibility in performing tasks. To achieve full control of an object in 3D space, a manipulator typically needs six DOF, which allow it to position and orient its end-effector (SICILIANO et al., 2008).

Kinematic models serve as a mathematical representation of the manipulator's structure and motion capabilities. As highlighted in the COMPAS framework documentation (RUST et al., 2018), these models are often described using the Unified Robot Description Format (URDF). The URDF defines the connections between links and joints, their geometric relationships, and additional elements like external axes, offering a tree-like structure

that mirrors the manipulator's physical design. The kinematic model not only facilitates accurate motion planning and simulation but also ensures that the manipulator operates within its defined workspace accessible by its end-effector. The interplay between the physical structure of a manipulator and its kinematic model underscores the importance of well-designed models in translating mechanical configurations into precise and reliable robotic actions.

In the domain of robotic manipulators, kinematic models serve as the foundation for understanding and controlling the relationship between a manipulator's physical structure and its motion capabilities. The COMPAS FAB framework provides a comprehensive set of classes and tools to define and manipulate these kinematic models, facilitating the design, simulation, and control of robotic systems with scientific rigor.

At the core of the COMPAS FAB framework is the `RobotModel` class, which encapsulates the structural and kinematic configuration of a robotic manipulator. This model is composed of `Link` objects, representing the rigid bodies that provide the manipulator's structural integrity, and `Joint` objects, which define the articulations between links. The `Joint` class supports various joint types, including revolute, prismatic, and fixed, reflecting the degrees of freedom (DOFs) necessary to achieve motion within the kinematic chain. This structure is consistent with the Unified Robot Description Format (URDF), ensuring interoperability and standardization across robotic software systems RUST et al., 2018.

3.3.2 Coordinate Frames and Transformations

In robotic systems, the definition and consistent use of coordinate frames are essential for ensuring seamless integration and reuse of robot drivers, models, and libraries. These frames establish shared conventions for spatial relationships and orientations, forming the foundation for planning and executing robotic fabrication processes. The World Coordinate Frame (WCF) serves as a global reference, with its origin fixed in space and its Z-axis oriented upwards, aligning with the ROS "map" convention. This frame is crucial for processes involving multiple robots operating in a shared workspace, robots with external axes, or mobile robots. By default, the WCF coincides with the Robot Coordinate Frame (RCF). The Robot Coordinate Frame (RCF), referred to as "base_link" in ROS, originates at the base of the robot and serves as the primary reference system for its mechanical structure. It is defined relative to the WCF, ensuring consistency in positioning and orientation. The Tool0 Coordinate Frame (T0CF) is anchored at the tip of the robot's last link, inheriting its relationship from the RCF. This frame represents the unmodified tool mount point and serves as the basis for defining the Tool Coordinate Frame (TCF). The TCF, often referred to as the Tool Center Point (TCP), represents the active working point of the tool, such as the tip of a gripper or welding nozzle. It is defined in relation to the T0CF, accounting for the specific tool geometry. Finally, the Object Coordinate Frame (OCF) describes the position and orientation of the work object or built structure in relation to the WCF. This frame is critical for defining the spatial relationship between the robot and its task environment, enabling precise manipulation and fabrication processes. Together,

these coordinate frames create a structured hierarchy, ensuring accurate transformations and efficient planning across diverse robotic applications (MELE and many OTHERS, 2017-2021).

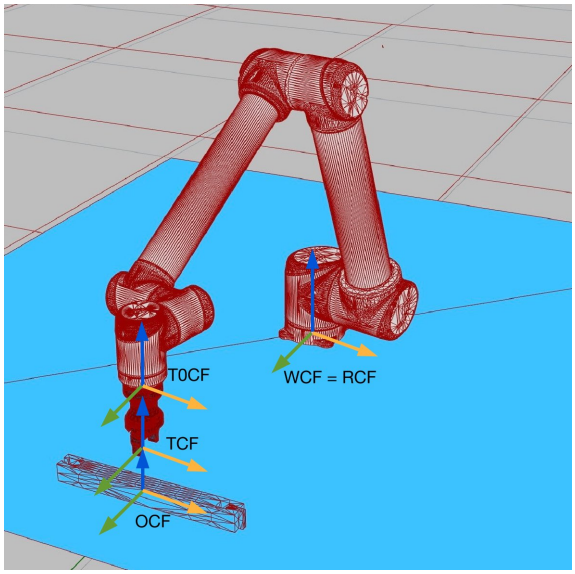


Figure 3.5: Coordinate Frames

In the COMPAS framework, the concepts of frames and transformations are fundamental tools for describing positions, orientations, and coordinate systems in 3D space. Frames are fundamental tools in robotics for describing one coordinate system relative to another, encompassing both position and orientation. A frame generalizes these concepts by combining a position vector and a rotation matrix. The position vector locates the point in space being described, while the rotation matrix defines the orientation of the coordinate system (CRAIG, 2021. The class `compas.geometry.Frame` is employed to describe and manage these coordinate systems, ensuring compatibility with standard robotics conventions. Frames can be used to define both local and global coordinate systems and provide methods to transform points and vectors between these systems. They also allow for describing and converting between said different conventions for pose orientation, such as Euler angles, axis-angle representation, and quaternions, commonly used in industrial robotics. The COMPAS framework supports conversions between these formats, ensuring compatibility and consistency in pose descriptions across diverse applications (MELE and many OTHERS, 2017-2021).

Transformations, on the other hand, are mathematical operations that modify geometric entities like frames, points, and vectors. The Transformation class serves as a base for various transformation types, including rotation, translation, scaling, reflection, projection, and shear. Transformations can represent complex operations, such as changing a frame's orientation to align with a desired pose (MELE and many OTHERS, 2017-2021).

3.3.3 Forward and Inverse Kinematics

Forward Kinematics

Forward kinematics is the process of determining the position and orientation of a robot's end-effector in Cartesian space, given the joint parameters such as angles or displacements. Using a series of transformations, typically modeled using the Denavit-Hartenberg (D-H) convention, forward kinematics maps the robot's joint space configuration to the global coordinate system. Each transformation describes the rotation and translation between adjacent links, and the overall position of the end-effector is obtained by multiplying these transformations sequentially along the robot's kinematic chain. Forward kinematics is foundational in robotics and serves as a prerequisite for motion planning and control, as described by CRAIG, 2021. In practice, frameworks like ROS compute forward kinematics using pre-defined kinematic models of robotic manipulators, enabling real-time calculations for tasks like path visualization in simulation tools such as RViz (ROS.ORG, n.d.-a). In COMPAS FAB, this process is facilitated by the `forward_kinematics()` function, which uses the `compas_fab.robots.Configuration` class to define the state of each joint in the robot's articulated body. A straightforward approach to computing forward kinematics leverages the robot model's properties without requiring a running ROS instance. For example, using the COMPAS FAB library, a RobotLibrary instance can calculate the world coordinate frame (WCF) of the end-effector given a specified joint configuration. The resulting pose specifies the end-effector's position and orientation in the world coordinate system. Additionally, if a robot is connected to a ROS client, the `forward_kinematics()` function can resolve the pose using ROS's kinematic solver. By utilizing the `compas_fab.backends.RosClient`, users can load the robot model, validate its identity, and compute the same results within a ROS environment. This dual functionality ensures flexibility, enabling forward kinematics calculations both in standalone setups and within integrated ROS-based workflows RUST et al., 2018.

Inverse Kinematics

Inverse kinematics, by contrast, solves the problem of determining the necessary joint configurations required to position the robot's end-effector at a desired location and orientation in Cartesian space. Unlike forward kinematics, which always yields a single deterministic solution, inverse kinematics is more complex due to the possibility of multiple solutions or even no solution for certain positions outside the robot's workspace. Analytical or numerical methods are used to solve the inverse kinematics equations, often relying on iterative approaches when closed-form solutions are not feasible. In "Introduction to Robotics: Mechanics and Control," CRAIG, 2021 highlights the challenges of inverse kinematics, including singularities and redundancy. ROS integrates these principles into motion planning algorithms within MoveIt!, allowing robotic systems to determine joint trajectories for user-defined end-effector goals in real or simulated environments (ROS.ORG, n.d.-a).

The `inverse_kinematics()` function in COMPAS FAB facilitates this computation by utilizing a predefined robot model and an initial guess for the joint states, such as the zero configuration. For example, given a target pose defined in the world coordinate frame (WCF), the function computes a valid joint configuration that enables the robot to achieve this pose within its kinematic constraints. Moreover, the functionality is extended functionality by allowing users to request multiple solutions through the `iter_inverse_kinematics()` function, which iteratively computes alternative joint configurations that satisfy the target pose. By specifying parameters such as the maximum number of results, this iterative approach provides greater flexibility in selecting configurations that optimize for criteria such as joint limits or collision avoidance. These computations rely on the integration of ROS, which provides access to advanced kinematic solvers, ensuring robustness and precision in addressing the inverse kinematics problem. Such capabilities are essential in applications involving robotic manipulation, where accurate and feasible motion planning is critical (RUST et al., 2018).

3.3.4 Motion Planning

Motion planning and trajectory planning are fundamental tasks in robotic control, essential for enabling robots to perform complex tasks autonomously. Motion planning involves determining a path that a robot must follow from its initial position to a target position while avoiding obstacles and ensuring safe navigation through the environment (SICILIANO et al., 2008). On the other hand, trajectory planning goes a step further by incorporating time-dependent considerations, such as velocity, acceleration, and deceleration, to ensure smooth and feasible movement (LAVALLE, 2006). These two aspects are often interconnected, as trajectory planning typically builds upon the solution obtained from motion planning, adding temporal elements to the path. In practice, motion planning can be achieved using algorithms like Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM), which are designed to explore complex and high-dimensional spaces (LAVALLE, 2006). The integration of both motion and trajectory planning is crucial for effective robotic operation, particularly when performing tasks that require precise positioning and smooth, efficient movements (SICILIANO et al., 2008).

Motion Planning in COMPAS FAB

Motion planning in the COMPAS FAB framework enables the computation of feasible paths and trajectories for robotic manipulators while avoiding collisions and satisfying task constraints. The process integrates with ROS and MoveIt to leverage their planning capabilities. Central to motion planning are the `Robot` class, which represents the robotic system, and the `PlanningScene` class, which models the environment and obstacles. Using the `plan_motion()` function, users can compute paths to move the robot from an initial to a target configuration. This function generates a joint-space trajectory that ensures the robot's kinematic feasibility.

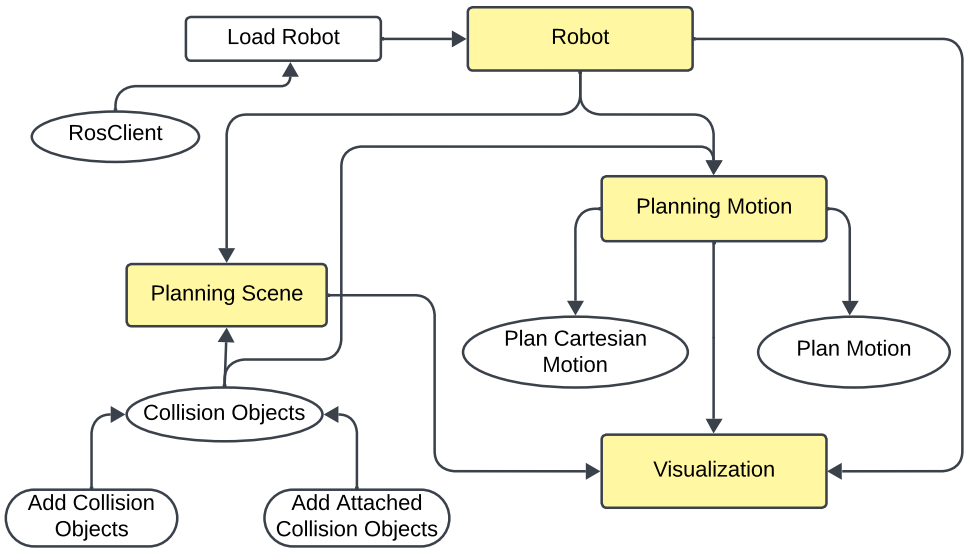


Figure 3.6: Relations Between Planning and Visualizing Robotic Motion

Cartesian motion planning is supported through the `plan_cartesian_motion()` function, which allows precise path planning for the robot's end-effector in Cartesian space. The user specifies waypoints as `Frame` objects, and the function calculates a trajectory that respects constraints such as obstacle avoidance and joint limits.

Once planned, trajectories can be visualized and validated using COMPAS FAB's simulation capabilities. The Trajectory class encapsulates the planned motion, and tools like `AttachedCollisionMesh` and `CollisionMesh` support integrating objects into the planning scene for enhanced collision checking. COMPAS FAB's API ensures flexibility in configuring and executing both joint-space and Cartesian-space motions, enabling solutions for complex robotic tasks (RUST et al., 2018).

3.3.5 Planning Scene and Collision

In robotic systems, scene planning and collision avoidance are fundamental components of motion planning, ensuring that a robot can operate safely and efficiently within its environment. According to SICILIANO et al., 2008, the planning scene encapsulates the geometric and spatial relationships between the robot and its workspace, including static and dynamic obstacles. This representation is essential for calculating valid configurations and trajectories that respect environmental constraints. Scene planning involves integrating obstacle information into the motion planning framework, ensuring that the robot's path avoids collisions and adheres to task-specific constraints, such as avoiding fragile objects or maintaining clearance from walls.

The concept of collision detection is rooted in kinematic and dynamic modeling, where the workspace and robot geometry are analyzed to identify potential intersections. As CRAIG, 2021 explained, collision-free planning requires precise modeling of the robot's links and joints in relation to the environment. This is achieved by defining bounding volumes or

meshes for obstacles and robot parts and performing collision checks during trajectory generation. Additionally, redundant degrees of freedom in manipulators can be leveraged to avoid obstacles while maintaining task accuracy. Together, scene planning and collision avoidance are indispensable for enabling robots to perform tasks in unstructured environments with both precision and safety.

Planning Scene and Collision in COMPAS FAB

In the COMPAS FAB framework, the `PlanningScene` class plays a pivotal role in representing the robot's environment, including static and dynamic obstacles, to facilitate collision-aware motion planning. The planning scene integrates the robot model with a detailed description of the surrounding workspace, enabling robust collision detection and avoidance during motion planning. Objects in the scene can be represented using `CollisionMesh` or `AttachedCollisionMesh`. A `CollisionMesh` defines a static obstacle with a mesh geometry, while an `AttachedCollisionMesh` allows users to model objects attached to the robot, such as tools or payloads, which move along with the manipulator.

Collision checking in COMPAS FAB is integral to the planning process and is handled during both configuration validation and trajectory computation. Functions like `add_collision_mesh` and `add_attached_collision_mesh` allow users to dynamically update the planning scene by adding or removing objects. The `is_configuration_valid` function checks whether a specific robot configuration avoids collisions, adheres to joint limits, and satisfies custom constraints.

The planning scene is critical for tasks requiring precise interaction with objects or environments, such as pick-and-place operations or constrained manipulations. By combining robust collision modeling and real-time updates, COMPAS FAB ensures that motion plans are feasible, safe, and optimized for the intended task (RUST et al., 2018).

3.3.6 URDF - Unified Robotic Description Format

URDF files are an XML-based format in robotics that describe a robot's structure, kinematics, and properties. They define links and joints, specify collision and visual geometries, and may include sensors and actuators, providing the data necessary for simulation, visualization, and motion planning within frameworks like ROS. Tools like Gazebo use URDF files to simulate robot-environment interactions, while motion planning libraries compute feasible trajectories, ensuring compatibility across components.

URDF also supports geometric information, such as collision meshes, and physical properties like inertia and joint limits. Although effective for constructing kinematic chains and complex robot configurations, URDF has limitations in encoding semantic information or advanced constraints, which are managed using formats like the Semantic Robot Description Format (SRDF). By integrating URDF models with tools like Gazebo for

simulation or RViz for visualization, developers can efficiently model robots and streamline application development (ROS.ORG, n.d.-a).

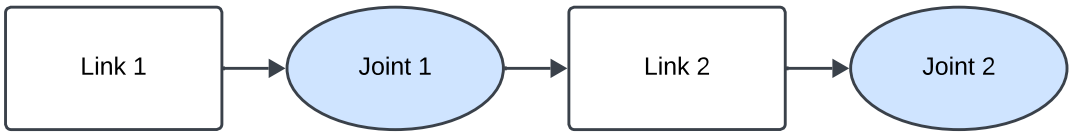


Figure 3.7: Link and Joint relation

The COMPAS Fab library integrates URDF files to facilitate the modeling, simulation, and control of robotic systems, particularly in the context of robotic fabrication. URDF files, being a standardized format for representing robot structure and kinematics, serve as an input for defining the robot's configuration within COMPAS Fab. Specifically, the library uses URDF files to establish the robot's link and joint structures, as well as to determine the kinematic chain necessary for motion planning and trajectory execution.

In COMPAS Fab, URDF files enable the import of detailed robotic models, which can be used to perform tasks such as motion planning, inverse kinematics, and collision detection. By converting the URDF model into a format compatible with the COMPAS framework, users can simulate the robot's behavior in a virtual environment, leveraging tools for path planning and visualization. Additionally, the URDF data structure ensures compatibility with various external tools, such as ROS-based simulators (e.g., Gazebo). As an alternative to Gazebo, Grasshopper, in conjunction with COMPAS Fab, provides a flexible parametric simulation environment for robotic systems, enabling users to visualize and interact with robotic models in a design-oriented context. This allows for real-time simulation, path optimization, and task execution within Grasshopper's parametric workflow.

Furthermore, COMPAS Fab extends the utility of URDF files by incorporating robot-specific details such as joint limits, sensor data, and end-effector specifications, which are critical for robotic fabrication tasks. The integration of URDF files allows for a seamless connection between the robot's digital model and real-world operations, facilitating efficient control and optimization of robotic systems in fabrication processes.

In summary, the COMPAS Fab library effectively utilizes URDF files to bridge the gap between robot design, simulation, and execution, ensuring accurate representation and control of robotic systems in the context of robotic fabrication and related applications. Grasshopper, as an alternative to Gazebo, serves as an integrated simulation environment within this ecosystem, enabling users to explore and optimize robotic workflows.

Chapter 4

Implementation

This chapter outlines the detailed process of implementing a modular timber joinery system and the integration of robotic simulation for the assembly tasks. It begins with an exploration of the modular timber joinery system, detailing the design intent and the functional requirements necessary for creating efficient timber components. Following that, the timber structure's design is presented, alongside the analysis of its assembly steps and structural integrity.

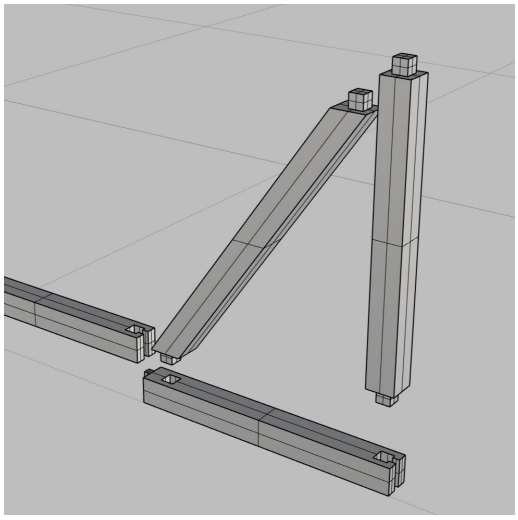
The chapter further progresses by discussing the setup and customization of a robotic simulation framework using COMPAS FAB within Grasshopper. This includes a comprehensive guide on configuring the Docker image for COMPAS FAB, setting up the backend GUI with XMing, and customizing the robot's URDF model. Moreover, a detailed explanation of the MoveIt! Configuration Package for the UR10 robot and adaptive gripper is provided, along with instructions on building a custom Docker image for the simulation environment.

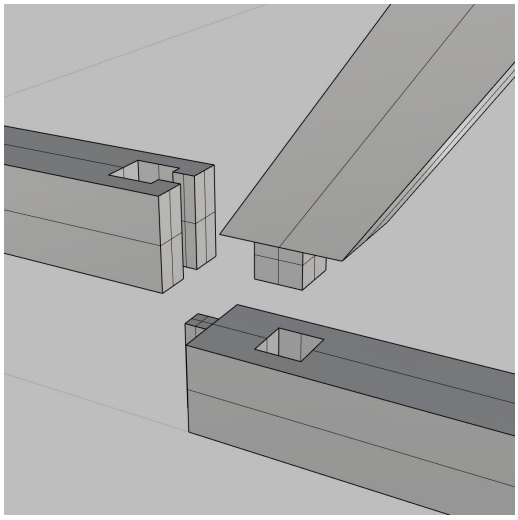
Additionally, the chapter covers the technical process of converting STEP files to JSON format for seamless structural data import, enhancing the integration between the timber design and robotic simulation. To facilitate user interaction with the robotic system, an interactive control interface using Grasshopper is introduced, allowing for real-time simulation of the robotic assembly tasks. Specific examples of robot workflows for pick-and-place cycles are given, followed by a discussion on the Grasshopper component blocks used for simulated robotic assembly operations.

Each section of this chapter contributes to the comprehensive development of a robotic system capable of efficiently handling modular timber joinery tasks, offering a step-by-step breakdown of both the design and implementation phases.

4.1 Modular Timber Joinery System

4.1.1 Design Intent and Functional Requirements





The parametric timber joinery system designed for robotic fabrication employs a plug-in connection mechanism to facilitate precise and tool-less assembly. The system consists of rectilinear timber components with customized interlocking joints, allowing for efficient and repeatable connections. The plug-in mechanism ensures initial alignment and stability, enabling components to hold together without the immediate need for fasteners or adhesives. This temporary stability provides a critical window for subsequent fixation using screws, either by a human operator or a secondary robotic system.

A key aspect of the design is the angled beam integration, which enhances its applicability in load-bearing structures such as trusses or frame assemblies. The precise geometric adaptation of the joint system ensures compatibility with automated fabrication workflows, allowing robotic arms to assemble components with high accuracy and minimal material waste (GRAMAZIO and KOHLER, 2008). This approach is particularly advantageous in mass customization, enabling the scalable production of complex timber structures while maintaining structural integrity and material efficiency.

This method aligns with current advancements in digital fabrication and computational design, offering an optimized workflow for sustainable construction. The modular nature of the system enhances its adaptability in reconfigurable architectural frameworks, where disassemblable and reusable components contribute to a circular material economy (GEISSDOERFER et al., 2017). Additionally, by integrating robotic assembly with temporary self-stabilization, the system reduces reliance on immediate fastening, further optimizing construction time and labor efficiency.

4.1.2 Assembly Steps

The robotic assembly of modular timber structures requires a well-defined sequence to ensure structural stability and fabrication efficiency. In this study, a parametric approach was employed to model the stepwise construction process, facilitating precise path planning for robotic execution. The assembly sequence was developed using computational design tools in Rhino and Grasshopper, enabling a systematic integration of horizontal, vertical, and diagonal timber components.

To achieve an optimized construction order, the structure was divided into discrete assembly steps, considering both mechanical constraints and robotic feasibility. Each step follows a logical progression, starting with foundational horizontal elements, followed by vertical and diagonal reinforcements, and concluding with upper-level horizontal components. This sequential strategy ensures proper load distribution and minimizes potential collisions during robotic handling.

The following sequence illustrates the stepwise assembly process of the timber structure, employing the modular joinery system.

1. Step: Initial Horizontal Component Placement (Figure 4.1)

The assembly process begins with the placement of primary horizontal timber members. These elements serve as the base framework, ensuring alignment and providing foundational support for subsequent structural elements.

2. Step: Integration of Vertical and Diagonal Components (Figure 4.2)

The second stage introduces vertical and diagonal members, which contribute to the structural integrity by forming triangulated load-bearing elements. These components are strategically positioned to enhance stability and resistance to lateral forces.

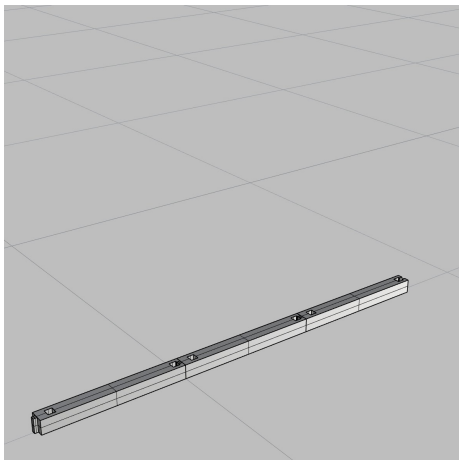


Figure 4.1: Horizontal Components

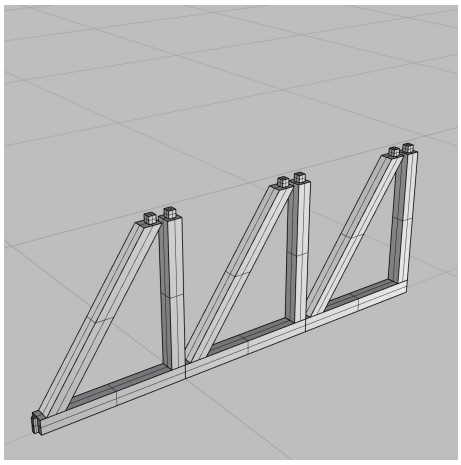


Figure 4.2: Vertical and diagonal Components

3. Step: Additional Horizontal Reinforcement (Figure 4.3)

Further horizontal elements are added, interlocking with existing components to reinforce the structure. This step increases the rigidity of the assembly and facilitates the seamless connection of upper structural elements.

4. Step: Final Vertical and Diagonal Member Placement (Figure 4.4)

A secondary set of vertical and diagonal components is incorporated, completing the primary framework. These elements optimize load distribution and enhance overall mechanical performance.

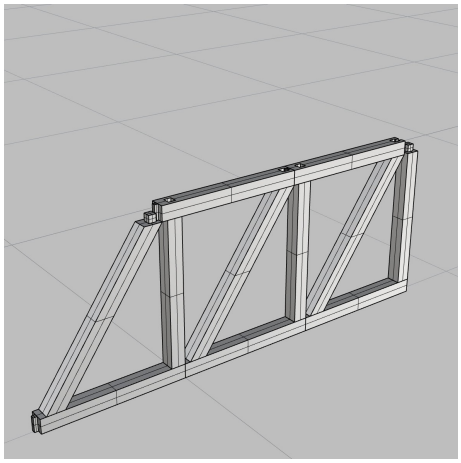


Figure 4.3: Horizontal Components

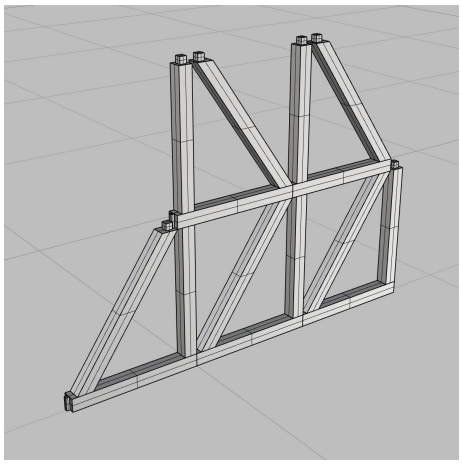


Figure 4.4: Vertical and diagonal Components

5. Step: Final Horizontal Component Integration (Figure 4.5)

The final step involves the placement of the uppermost horizontal members, ensuring full structural continuity. This stage finalizes the modular system, demonstrating the effectiveness of the interlocking joinery approach in constructing prefabricated timber structures.

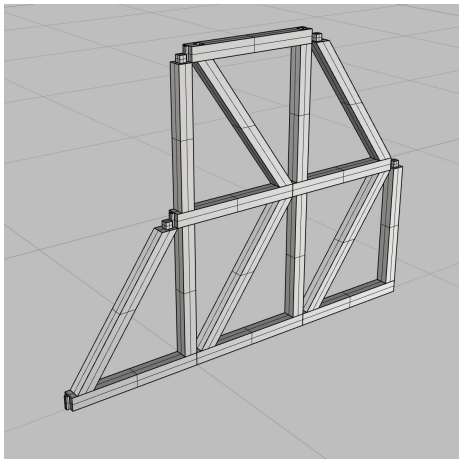


Figure 4.5: Horizontal Components

This sequential process emphasizes the efficiency and adaptability of modular timber construction. The integration of computational design and digital fabrication enables precise assembly and improving structural performance.

4.1.3 Design and Parameters of the Timber Components

Horizontal Timber Component

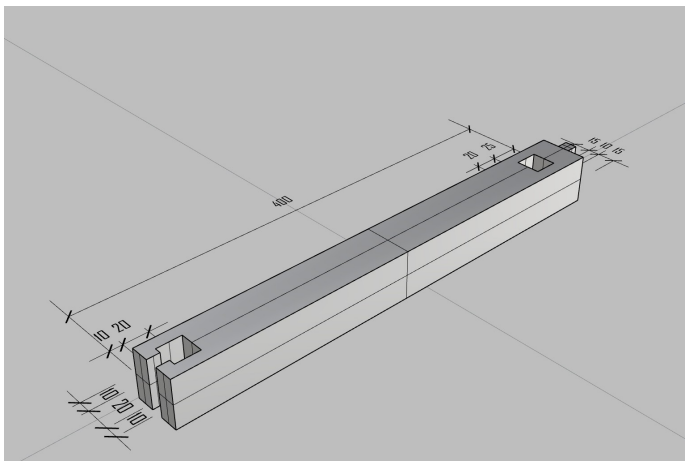


Figure 4.6: Perspective View of Horizontal Component in Millimeters

The horizontal timber component measures 400 mm in length and is designed with defined interlocking joints. The component features a symmetrical profile with detailed geometric specifications for accurate assembly.

At one end, the component incorporates a double tenon slot, the smaller tenon slot measuring 10 mm in width and 10 mm in depth, facilitating interlocking with horizontal elements and the bigger tenon slot measuring 20mm by 20mm for its interlocking with vertical elements. The opposite end features a rectangular tenon measuring 10 mm × 10 mm, accompanied by a 20mm x 20mm longitudinal slot for interlocking diagonal elements

Vertical Component

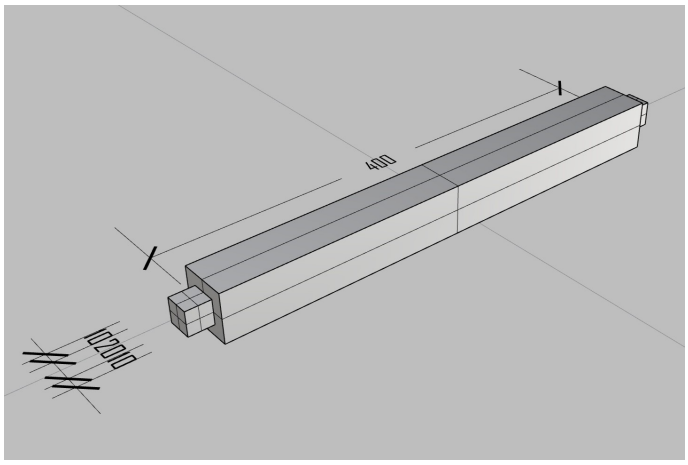


Figure 4.7: Perspective View of Vertical Component in Millimeters

The vertical timber component has a cubic tenon measuring 20 mm per side at both ends. The tenons are designed for precise alignment and insertion into a corresponding recess. The component is inserted vertically at an angle of 90 degrees into the horizontal counterpart of the structure.

Diagonal Component

The diagonal timber element features a cubic tenon, each side measuring 20 mm, at both ends. This element is angled at 55 degrees as it is fitted into the horizontal part of the structure.

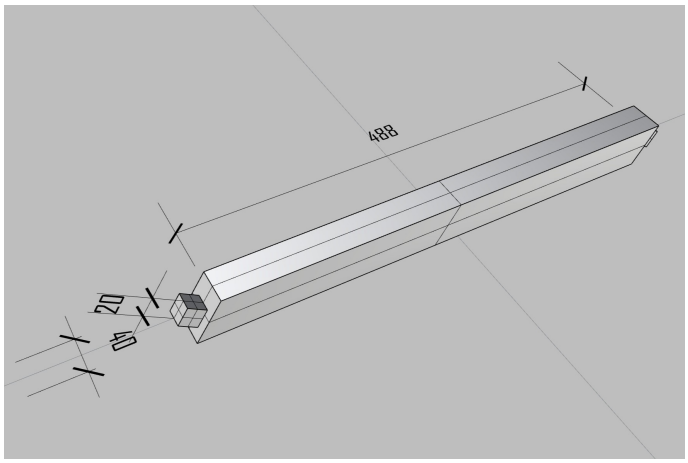


Figure 4.8: Perspective View of Diagonal Component in Millimeters

This component plays a crucial role in load distribution by transferring vertical loads into axial forces, reducing bending moments on vertical and horizontal members. It enhances structural stability by resisting lateral forces, preventing deformation under external loads

such as wind or seismic activity. Additionally, the diagonal beam optimizes compression and tension forces, ensuring efficient weight distribution while minimizing material usage. This improves overall rigidity and strength, making the structure more resilient and efficient.

The structured joinery details ensure compatibility with other components and therefore a stable connection within a predefined architectural or structural configuration.

Manufacturing of the Timber Components

To manufacture the custom timber components, various machining techniques could be employed, combining traditional methods with modern technologies to achieve high precision and customization. CNC routers and milling machines would be key to creating the complex joints required for the components. These systems allow for repeatable, accurate cuts, enabling the production of tailored components while minimizing material waste. The use of CNC machines would facilitate the efficient creation of joints that are precise and consistent across all units.

For the fabrication of mortise and tenon joints, CNC milling machines could be used to shape the components, while specialized mortisers would be employed to cut square or rectangular holes. Mortisers, designed specifically for such tasks, would ensure the mortises are accurately formed, allowing for proper fitting of the tenons. This combination of tools would provide a means to achieve strong, precise connections in the timber components, contributing to the overall structural integrity.

By employing these techniques, it would be possible to manufacture custom timber components with a high degree of precision and repeatability, making them suitable for large-scale production as well as bespoke applications. The 2D drawings of the components can be found in the appendix for further reference (A.6).

4.2 Setup and Configuration of UR10 Robot with Hand-E Gripper

4.2.1 Docker Integration for Custom Configuration

To use COMPAS Fab with a UR10 robotic arm equipped with a Robotiq Hand-E adaptive gripper as its end-effector, several preparatory steps are required. Specifically, a Docker image must be created with a MoveIt configuration tailored to this setup. Since a pre-fabricated URDF file for this combination does not exist, it is necessary to create a custom URDF file. The tools needed for this, are Docker Desktop and an IDE (Integrated Development Environment), e.g. Visual Studio Code. The following steps, which are also visualized in figure (4.9), outline the process of creating a custom Docker image with all necessary configurations:

1. Create a Custom URDF:
   A URDF file is developed, that defines the kinematic structure of the UR10 robot model and includes the Robotiq Hand-E gripper as its end-effector. This involves specifying all links, joints, and the end-effector's attachment to the robot arm and ensures, that the URDF file accurately reflects the physical configuration and parameters of the system.

2. Generate a MoveIt! Configuration:
   The custom URDF file can now be used to create a MoveIt configuration package. This step involves setting up the robot's planning groups, collision geometries, joint limits, and motion planning settings to enable COMPAS Fab to simulate and control the UR10 with the attached gripper.

3. Create a copy of docker image to safe for further use:
   Now, a new copy of the docker image with the new MoveIt! Configuration package can be created and run.

These steps ensure that the custom Docker image includes all the necessary components and configurations for effective integration and operation of the UR10 and the Hand-E gripper within the COMPAS Fab framework.

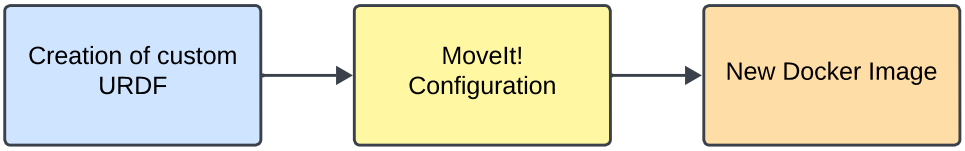


Figure 4.9: Workflow for Customized Robotic Simulation Environment

The setup for the Docker image with the custom MoveIt configuration is closely aligned with the guidelines provided in the COMPAS FAB documentation. COMPAS FAB offers a robust foundation for configuring robotic systems by integrating MoveIt for motion planning and simulation. However, in the case of the UR10 robotic arm equipped with the Robotiq Hand-E adaptive gripper, additional adjustments are necessary to enable control of the gripper fingers.

Specifically, while the standard COMPAS FAB documentation outlines the configuration of a robotic manipulator for motion planning, it does not inherently account for the actuation of end-effector components, such as gripper fingers. To address this, the custom MoveIt configuration package must include additional joint definitions and control interfaces for the gripper. This involves extending the URDF file to include the gripper's kinematics and integrating these changes into the MoveIt configuration to enable finger-specific motion planning. Furthermore, the Docker image must incorporate these modifications to ensure that the simulation and control environment accurately reflects the full range of robotic capabilities, including coordinated movements of both the arm and the gripper.

4.2.2 Backend GUI - XMing

Visualization forwarding display enables the direct forwarding of graphical interfaces from remote systems to a local operating system. This is achieved by forwarding the X11 display protocol. However, full platform compatibility is not always guaranteed. To use this feature, the X11 server XMing is installed for Windows. Next, X11 security is configured by adding the machine's IP address to the file "%ProgramFiles(x86)%\XMing\X0.hosts" on Windows (with administrative privileges). Finally, the DISPLAY environment variable is set to point to the X11 server and for Docker environments, DISPLAY=host.docker.internal:0.0 is added to the Docker configuration inside the docker-compose.yml file, which will be explained further later on (RUST et al., 2018).

4.2.3 Custom URDF for UR10 with end-effector

Adaptive Gripper Hand E Meshes

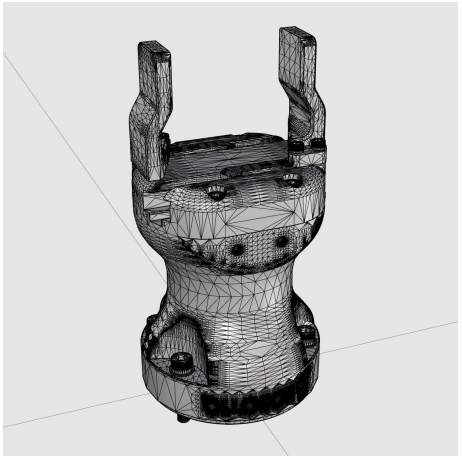


Figure 4.10: Robotiq Hand E

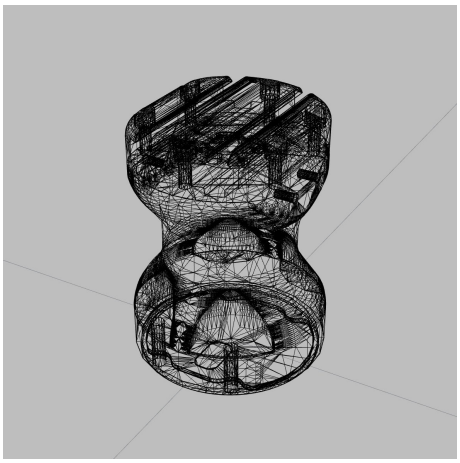


Figure 4.11: hand_e_base.stl

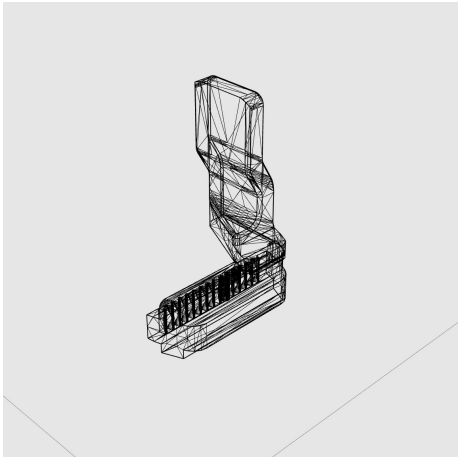


Figure 4.12: hand_e_finger_1.stl

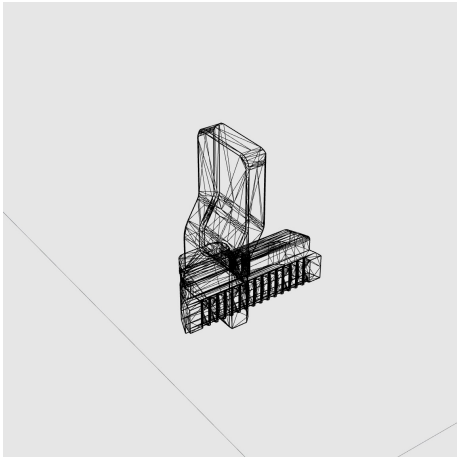


Figure 4.13: hand_e_finger_2.stl

Before initiating the creation process, the mesh files for the gripper components must be generated in the .stl format. The required .step files for the gripper can be obtained from the Robotiq website (ROBOTIQ, 2023). These files should be opened in a CAD program

capable of handling .step files, where the meshes can be reduced and converted to the .stl format. It is essential to save each mesh component of the gripper—specifically the base, finger one, and finger two—as separate .stl files. This separation is necessary to ensure the independent movement of the gripper fingers, enabling them to open and close during operation. The base mesh, and the fingers accrodingly to the base position, should be exported centered at the origin of the world coordinate system.

New Dummy Docker Image

To configure the Docker environment for the UR10 robot using COMPAS FAB, the provided Docker image must first be downloaded from the COMPAS FAB documentation (RUST et al., 2018). The image, configured for the UR10 robot, requires a minor modification to enable the use of a Linux command prompt within the container. The configuration file docker-compose.yml should be opened in a text editor, such as Visual Studio Code, and the following lines added under the command section within the ros-core configuration:

```
stdin_open: true
tty: true
```

This adjustment ensures interactive command-line functionality within the Docker container. After making this change, Docker Desktop must be started, and the following command executed to launch the container:

```
docker-compose up -d
```

Once the container is running, the file system of the ros-core container can be accessed via Docker Desktop, revealing the typical structure of a Linux environment. Within the file system, the directory /root/catkin_ws/src serves as the workspace for further configurations.

To open a Linux command prompt inside the Docker container, the following command is executed in the terminal or Visual Studio Code:

```
docker-compose exec ros-core bash
```

Within the newly opened command prompt, the next step involves creating a new Catkin package.

Catkin Workspace

In the opened command-line interface, a new Catkin package is created. Catkin is the build system utilized in ROS to manage dependencies, organize packages, and build

projects (ROS.ORG, n.d.-a). It establishes a structured workspace and maintains the necessary files for compiling and executing ROS packages.

The following command is used to create a new package named ur10_hand_e:

```
catkin_create_pkg ur10_hand_e
```

This command generates a folder named ur10_hand_e within the /catkin_ws/src directory. Verification of the creation can be performed by examining the directory. The newly created folder includes a file named package.xml, which contains metadata and dependencies for the package. The package.xml file must be edited, and the following line added after the existing `<buildtool_depend>catkin</buildtool_depend>`:

```
1  <buildtool_depend>catkin</buildtool_depend>
2  <test_depend>roslaunch</test_depend>
3  <build_export_depend>joint_state_publisher</build_export_depend>
4  <build_export_depend>robot_state_publisher</build_export_depend>
5  <build_export_depend>rviz</build_export_depend>
6  <build_export_depend>xacro</build_export_depend>
7  <exec_depend>joint_state_publisher</exec_depend>
8  <exec_depend>robot_state_publisher</exec_depend>
9  <exec_depend>rviz</exec_depend>
10 <exec_depend>xacro</exec_depend>
```

To proceed with the configuration, additional folders must be created within the ur10_hand_e directory. Specifically, a folder named meshes is required, which should contain two subfolders: collision and visual. The three .stl CAD files corresponding to the gripper must be copied into both the collision and visual subfolders. Up to this step, the process closely aligns with the guidelines provided in the COMPAS FAB documentation.

Furthermore, three additional folders named launch, rviz, and urdf must be created within the ur10_hand_e directory. These folders will serve as repositories for the necessary configuration and launch files required for integrating the UR10 robot and the Hand-E gripper into the simulation and visualization environment.

Creating Xacro and URDF files

To create the configuration for the Robotiq Hand-E gripper, a new text file is generated using a text editor such as Notepad and renamed with the extension .xacro, resulting in a file named hand_e.xacro. The .xacro format stands for XML Macros, which is an extension of the URDF (Unified Robot Description Format) that allows for modularity and reuse of robot model components. Xacro files provide a mechanism to include parameters, conditional statements, and macros, simplifying the creation and maintenance of complex robotic models. They are particularly useful for robots with modular configurations or when multiple similar components are required (ROS.ORG, n.d.-a).

In this case, the hand_e.xacro, A.1, file will contain the specific description of the Robotiq Hand-E gripper, including its kinematic properties, visual and collision meshes, and joint configurations. The file accepts two parameters: prefix, which allows for unique naming of components, and connected_to, which specifies the parent link to which the gripper is attached. The gripper's kinematic structure begins with a fixed joint (hand_e_base_joint), attaching the gripper base (hand_e_base) rigidly to the parent link. The <origin> tag specifies the joint's position and orientation relative to the parent.

The gripper base is defined as a <link> element with associated visual and collision meshes stored in the meshes folder. These meshes describe the physical appearance and collision geometry of the base. The gripper includes two prismatic joints, hand_e_finger_1_joint and hand_e_finger_2_joint, allowing linear motion for the gripper fingers along specified axes. These joints are configured with motion limits, including effort, velocity, and range constraints. The second finger joint mimics the movement of the first using the <mimic> tag, ensuring synchronized motion.

Finally, a tcp_joint connects the gripper to a tool center point (TCP) link, providing a reference frame for task planning. By encapsulating the gripper's kinematic and geometric definitions in a macro, the file enables efficient integration and reuse in robotic models, particularly for applications involving the UR10 robotic arm.

The second Xacro file is needed to connect the Robotiq Hand E gripper to the UR10 robotic arm. The root <robot> tag specifies the model's name, ur10_hand_e, and includes references to the two additional Xacro files: the UR10 robot description (ur10.urdf.xacro) and the Hand E gripper description (hand_e.xacro).

The UR10 robot model is included using the <xacro:ur10_robot> tag. Parameters such as prefix (set as an empty string) and joint_limited (set to true) configure the robot's namespace and enable joint limits to constrain its motion during simulation. Similarly, the hand E gripper is included via <xacro:hand_e> with the connected_to parameter set to tool0, indicating its attachment point on the robot. This modular inclusion facilitates reusability and adaptability for different configurations.

A fixed joint named world_joint connects the robot's base link (base_link) to the world coordinate frame (world). The <origin> tag specifies the position and orientation of the robot's base in the global coordinate system, here set to the origin with no rotation. This joint ensures that the robot's position is anchored in the simulation environment.

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="https://ros.org/wiki/xacro" name="ur10_hand_e">

  <!-- ur10 -->
  <xacro:include filename="$(find ur_description)/urdf/ur10.urdf.xacro" />
  <!-- end-effector -->
  <xacro:include filename="hand_e.xacro" />

  <!-- ur10 -->
  <!-- The ur10 xacro must be included with passing parameters -->
  <xacro:ur10_robot prefix="" />
  <!-- end-effector -->
  <!-- Here we include the end-effector by setting the parameters -->
  <xacro:hand_e prefix="" connected_to="tool0"/>

  <!-- define the ur10's position and orientation in the world coordinate system -->
  <link name="world" />
  <joint name="world_joint" type="fixed">
    <parent link="world" />
    <child link="base_link" /> <!-- TODO: check base_link name of robot -->
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
  </joint>
</robot>
```

Ensure that both Xacro files (ur10_hand_e.xacro and hand_e.xacro) are uploaded into the catkin_ws/src/ur10_hand_e/urdf folder within the Docker environment. The Xacro file must now be processed to generate a corresponding URDF file. To achieve this, navigate to the urdf folder within the workspace and execute the following command. This command utilizes the Xacro package in ROS to expand and evaluate the macros defined in the input Xacro file (ur10_hand_e.xacro) and produce the output URDF file (ur10_hand_e.urdf):

```
rosrun xacro xacro --inorder -o ur10_hand_e.urdf ur10_hand_e.xacro
```

The newly created urdf file can be checked with the following command:

```
check_urdf ur10_hand_e.urdf
```

The output looks as follows:

```
robot name is: ur10_hand_e
---------- Successfully Parsed XML ---------------
root Link: world has 1 child(ren)
    child(1):  base_link
        child(1):  base
        child(2):  shoulder_link
            child(1):  upper_arm_link
                child(1):  forearm_link
                    child(1):  wrist_1_link
                        child(1):  wrist_2_link
                            child(1):  wrist_3_link
                                child(1):  ee_link
                                child(2):  tool0
                                    child(1):  hand_e_base
                                        child(1): hand_e_finger_1
                                        child(2): hand_e_finger_2
                                        child(3): tcp
```

The output shows the structured kinematic chain of the ur10 robotic manipulator with an attached robotiq hand e gripper, starting from the base and extending through the shoulder, arm, and wrist joints to the end-effector. The end-effector includes a gripper mechanism with two fingers and a Tool Center Point (TCP) for precise measurement. Additionally, tool0 serves as the mounting point for external tools. This hierarchy defines the robot's movement and functionality, enabling both object manipulation and measurement tasks. For a detailed breakdown of the structure, refer to Figure (4.14).

The Custom URDF

The URDF file, shown as a hierarchical diagram in figure 4.14, represents a robotic system comprising a Universal Robots UR10 robotic arm equipped with a Robotiq Hand-E adaptive gripper as its end-effector. The kinematic chain starts with the base_link, which is fixed to the world coordinate frame via the world_joint and base_fixed_joint. This serves as the foundation for the articulated structure of the robot.

The robotic arm itself is modeled as a series of rigid links connected by revolute joints, enabling its six degrees of freedom. The shoulder_link is attached to the base_link via the shoulder_pan_joint, followed by the upper_arm_link, which is connected through the shoulder_lift_joint. The chain continues with the forearm_link, linked via the elbow_joint, and concludes with the wrist assembly: wrist_1_link, wrist_2_link, and wrist_3_link, connected by the wrist_1_joint, wrist_2_joint, and wrist_3_joint, respectively. Together, these links and joints form the standard configuration of the UR10 arm, providing its full range of motion and flexibility.

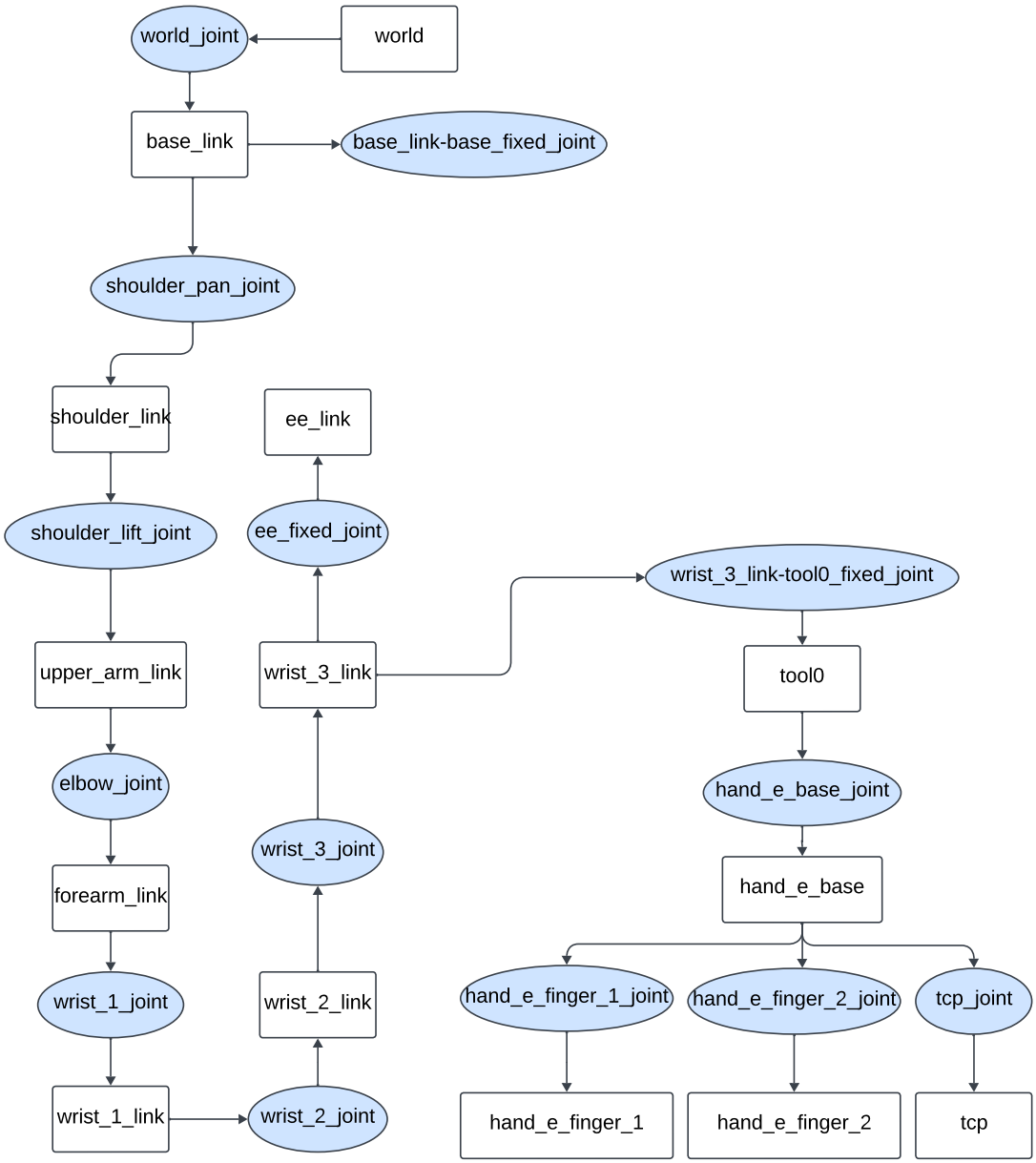


Figure 4.14: URDF Hierarchical Structure of UR10 with Hand E adaptive Gripper attached

At the end of the robotic arm, the Robotiq Hand-E adaptive gripper is attached. The gripper base (hand_e_base) is fixed to the wrist_3_link via the wrist_3_link-tool0_fixed_joint. The gripper's two fingers, represented as hand_e_finger_1 and hand_e_finger_2, are articulated through hand_e_finger_1_joint and hand_e_finger_2_joint, enabling adaptive gripping capabilities for objects of various shapes and sizes. The end-effector reference frame, or tool center point (tcp), is attached via the tcp_joint, providing a consistent coordinate system for task planning and execution.

This URDF structure captures the physical and kinematic details of the UR10 robotic arm and the Robotiq Hand-E gripper, enabling their use in simulation, motion planning, and real-world applications requiring precise manipulation and control.

Visualizing the URDF in RViz

After cloning the needed repository urdf_tutorial with the following command,

```
git clone https://github.com/ros/urdf_tutorial.git
```

the files located within the 'launch' and 'rviz' directories, display.launch and urdf.rviz, are copied into the corresponding 'launch' and 'rviz' directories in the urdf folder of the ur10_hand_e folder that had been established in a prior step, with these commands:

```
roscd urdf_tutorial
cp rviz/urdf.rviz ~/catkin_ws/src/ur10_hand_e/rviz/
cp launch/display.launch ~/catkin_ws/src/ur10_hand_e/launch/
cd ~/catkin_ws
```

Next, the 'display.launch' file must be opened inside Docker Desktop and modified to match the following configuration:

```
<launch>

  <arg name="model" default="$(find ur10_hand_e)/urdf/ur10_hand_e.urdf"/>
  <arg name="gui" default="true" />
  <arg name="rvizconfig" default="$(find ur10_hand_e)/rviz/urdf.rviz" />

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="
    joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true" />

    </launch>
```

The last step is to build the ROS (Robot Operating System) workspace by compiling the necessary packages and finally, the source devel/setup.bash command is run to configure the environment by sourcing the setup script, thereby setting up the necessary environment variables for the ROS workspace.

```
cd ~/catkin_ws
catkin_make
source devel/setup.bash
```

4.2.4 MoveIt! Configuration

A MoveIt configuration package is essential for integrating a robotic system into the MoveIt motion planning framework. This package comprises a set of configuration files that define

the robot's kinematic structure, planning groups, end effectors, and parameters crucial for motion planning, collision detection, and trajectory execution. Central to the configuration package is the Semantic Robot Description Format (SRDF) file, which extends the robot's Universal Robot Description Format (URDF) by incorporating semantic information such as joint groupings, default states, and self-collision matrices. Collectively, these files provide the necessary data for MoveIt's motion planning pipeline to function effectively (ROS.ORG, n.d.-a).

The MoveIt! Setup Assistant

The MoveIt Setup Assistant is a graphical tool designed to facilitate the creation of a MoveIt configuration package. By loading a robot's URDF file into the Setup Assistant, users can interactively define the robot's kinematic parameters, specify planning groups, assign end effectors, and generate a self-collision matrix. The tool also aids in defining joint limits and virtual joints, ensuring the robot's compatibility with the MoveIt motion planning framework. Upon completion of the configuration process, the Setup Assistant generates the SRDF file and other essential files, which are then organized into a configuration package ready for deployment in motion planning and simulation tasks. The following steps closely align with the MoveIt! documentation (PICKNIKINC., n.d.).

This method streamlines the integration of complex robotic systems into the MoveIt ecosystem, enabling precise and efficient motion planning for a wide range of robotic applications. Additional details and tutorials on using the MoveIt Setup Assistant can be found in the official ROS documentation (ROS.ORG, n.d.-a).

The MoveIt! Setup Assistant is initiated by executing the following command in the Linux terminal:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

This command launches the application, enabling the creation of a new MoveIt! configuration package. After selecting the "Create New MoveIt Configuration Package" option, the robot's URDF file, ur10_hand_e.urdf, is loaded to initialize the configuration process. This step establishes the robot's model as the basis for subsequent parameterization.

The self-collision matrix is a key component that identifies potential self-collisions within the robot's kinematic structure. In the "Self-Collisions" pane, the "Generate Collision Matrix" function performs a systematic check of possible collisions within the defined range of motion. Using a default sampling density of 10,000 checks, the matrix ensures that self-collisions are accounted for during motion planning, thus enhancing the safety and reliability of the robot's operation.

Virtual joints connect the robot's base link to the world coordinate frame, anchoring the robot within the simulation environment. These joints are defined in the "Virtual Joints" pane by specifying the joint name as ‘virtual_joint‘, setting the child link as ‘base_link‘,

and the parent frame as ‘world‘. A fixed joint type is typically used to represent a rigid connection. This setup ensures that the robot’s position and orientation are correctly established in the global reference frame.

Planning groups represent logical divisions of the robot, such as manipulators or end-effectors, used for motion planning. These groups are defined in the "Planning Groups" pane by specifying their names, kinematic solvers, and associated links or chains. The manipulator group uses ‘ur_kinematics/UR10KinematicsPlugin‘ as its kinematics solver and also has a kinematic chain. This needs to be added with ‘base_link‘ as the base link and ‘tcp‘ as the tip link. An Additional group, the end-effector group called ’gripper’, is defined by including links specific to the end-effector,‘hand_e‘, based on its corresponding Xacro file. These links need to be added, by choosing the following links in this order: hand_e_base, hand_e_finger_1, hand_e_finger_2, tcp. Also, a joint has to be added, by selecting the hand_e_finger_2_joint.

Robot poses are predefined joint configurations used as reference points during motion planning. These poses are defined in the "Robot Poses" pane by naming the pose (e.g., ‘home_pose‘) and specifying the corresponding joint values. The predefined poses serve as initial or target configurations for motion planning tasks, facilitating consistent and efficient operation. For this research purpose, no robot poses are defined at this point.

End-effectors are defined within the "End Effectors" pane by associating a planning group with its functional role. The gripper group can be designated as an end-effector by assigning it to the parent link ‘tool0‘. This designation establishes the role of the end-effector within the motion planning pipeline. The name of the end-effector of group ’gripper’ is called ’hand_e’.

The Controllers Pane in the MoveIt! Setup Assistant is used to configure a robot’s hardware controllers, allowing it to execute planned trajectories. Controllers are defined by their type, the joints or planning groups they manage, and other parameters. This ensures compatibility between the motion planning framework and the robot’s actuators (PICKNIKINC., n.d.). The button ’Auto Add FollowJointsTrajctory Controllers For Each Planning Group’ can be used to generate the needed controllers.

The Author Information pane allows the inclusion of metadata about the creator of the configuration package. This information is stored in the package metadata and is useful for documentation and future reference.

The process of generating a MoveIt! configuration package involves creating the necessary files and configurations to integrate a robotic system into the MoveIt! motion planning framework. The generation process begins with selecting a directory to store the configuration package. A dedicated folder is created to contain the package files, named according to the robot or project: ur10_hand_e_moveit_config. Within this directory, the Setup Assistant generates a series of files, including the Semantic Robot Description Format (SRDF) file, kinematics configurations, collision matrix data, and ROS launch files. These components are critical for motion planning, visualization, and execution in MoveIt!.

Once the package is generated, it is compiled within a ROS workspace using catkin_make. This step prepares the workspace for execution by building the configuration files and linking them with the required ROS dependencies. After compilation, the workspace is sourced to ensure that the generated package is accessible for further use.

The configuration can be tested by launching a demonstration file, typically included in the package, which initializes RViz and allows for the visualization and testing of the robot model within the MoveIt! framework. This structured process provides a robust foundation for integrating robotic systems into motion planning and simulation workflows.

```
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch ur10_hand_e_moveit_config demo.launch rviz_tutorial:=true
```

4.2.5 Custom Docker Image

To utilize the previously created configuration, it is necessary to copy the Docker image and rerun it with the updated settings. Access the ros-core container in Docker Desktop and retrieve the container ID. This ID is required to create a new Docker image based on the current state of the container.

The creation of the new image, named ur10_hand_e_image, is performed using the following command. Note that this command is executed in the local command prompt rather than the Linux environment within the Docker container. Replace "container_id" in the command with the actual container ID copied earlier:

```
docker commit container_id ur10_hand_e_image
```

Once this command is executed, the new image should appear in the "Images" tab of Docker Desktop, ready for further use. This process ensures that the updated configuration is preserved and can be rerun as needed.

The docker-compose.yml file should be edited to look like the following:

```
1  version: '2'
2  services:
3
4    moveit-demo:
5      image: ur10_hand_e_image
6      container_name: moveit
7      environment:
8        - ROS_HOSTNAME=moveit-demo
9        - ROS_MASTER_URI=http://ros-core:11311
10       - DISPLAY=host.docker.internal:0.0
11     depends_on:
12       - ros-core
```

```
    command:
      - roslaunch
      - --wait
      - ur10_hand_e_moveit_config
      - demo.launch
      - use_rviz:=true

  ros-core:
    image: ur10_hand_e_image
    container_name: ros-core
    ports:
      - "11311:11311"
    command:
      - roscore
    stdin_open: true
    tty: true

  ros-bridge:
    image: ur10_hand_e_image
    container_name: ros-bridge
    environment:
      - "ROS_HOSTNAME=ros-bridge"
      - "ROS_MASTER_URI=http://ros-core:11311"
    ports:
      - "9090:9090"
    depends_on:
      - ros-core
    command:
      - roslaunch
      - --wait
      - rosbridge_server
      - rosbridge_websocket.launch

  ros-fileserver:
    image: ur10_hand_e_image
    container_name: ros-fileserver
    environment:
      - ROS_HOSTNAME=ros-fileserver
      - ROS_MASTER_URI=http://ros-core:11311
    depends_on:
      - ros-core
    command:
      - roslaunch
      - --wait
      - file_server
      - file_server.launch
```

Once the docker-compose.yml file is configured, the environment can be deployed using again the docker-compose up command. This establishes a connection to the ROS system, allowing the robot to be integrated and loaded inside the Grasshopper environment.

4.3 STEP to JSON Conversion for Structural Data Import

To facilitate the robotic assembly of a modular timber structure, it is essential to extract geometric and spatial data from the digital model and format it for computational processing. The timber structure is initially designed in Rhinoceros 3D (Rhino) using custom parametric components and then exported as a STEP (.step) file, a standardized format for exchanging 3D geometry data. Since Grasshopper, requires structured data to manipulate and simulate the robotic assembly task, the STEP file must be processed and converted into a JSON format. This transformation extracts essential spatial properties, specifically the center of mass and orientation of each timber component, enabling precise robotic handling.

To achieve this, the conversion process is implemented using pythonocc, a Python wrapper for the Open Cascade Technology (OCC) geometric modeling kernel, which provides advanced tools for handling STEP files, solid geometry, and computational properties (CASCADE, 2023). The extract_coordinates_from_step function, A.2, first loads the STEP file using the STEPControl_Reader module and extracts its root shape representation. It then iterates through all solid elements in the file using TopExp_Explorer, identifying individual timber components. For each component, the script calculates the center of mass using the GProp_GProps class, normalizing and storing the computed coordinates. Additionally, the inertia matrix is obtained using MatrixOfInertia, from which the principal axes (x, y, z) are extracted to define the orientation of the component in space (pythonOCC COMMUNITY, 2023). The extracted component properties are then sorted by height to maintain the correct assembly sequence and stored in a structured JSON file. Each entry in the JSON output contains the center of mass coordinates and principal orientation axes, which can be directly imported into Grasshopper.

This JSON format allows the structured data to be used for robotic pick-and-place path planning and collision-free assembly sequencing within the Grasshopper environment. By leveraging pythonocc for computational geometry processing, the workflow enables seamless data exchange between CAD modeling and robotic simulation, ensuring that the assembly process is both precise and efficient. The integration of this data into Grasshopper allows for direct interaction with parametric design workflows and robotic simulation tools, supporting an automated and flexible fabrication process (R. MCNEEL, n.d.).

A key advantage of this approach is that any structure composed of the custom modular timber components within range of the robotic arm can be imported and processed in the same way. This means that designers are not limited to a fixed set of predefined assemblies but can freely experiment with different structural configurations in Rhino. Once the digital model is exported and converted, the robotic system can autonomously interpret and execute the assembly, allowing complete flexibility in fabrication. Whether constructing a simple frame or a complex spatial structure, the robot can use the same

automated workflow to assemble virtually any configuration designed within the modular system.

4.4 Grasshopper Playground as an Interactive Control Interface for Robotic Simulation with COMPAS FAB

The Grasshopper Playground, A.3, serves as an interactive interface for remotely controlling a simulated robotic system, with the potential for future integration into real-world robotic applications. While the ultimate objective is to achieve full automation—where the robot autonomously executes all necessary steps without direct user intervention—the development of a remote control framework provides a crucial intermediary stage for research, validation, and process planning.

A key objective of this research is to assess the capabilities and limitations of the COMPAS FAB library in the context of robotic fabrication. COMPAS FAB provides a set of preconfigured Grasshopper components that encapsulate functional Python code, simplifying key robotic operations such as loading the robot model, establishing a connection to a ROS client, and visualizing planned trajectories (RUST et al., 2018). These ready-made blocks allow users to interact with robotic systems without having to manually implement low-level communication protocols or motion planning algorithms, making Grasshopper a powerful interface for exploring robotic workflows.

Furthermore, the interactive nature of the Grasshopper Playground allows researchers and engineers to explore the feasibility of different robotic processes before full automation is implemented. In early-stage process planning, where task objectives may not yet be well defined, this approach enables iterative refinement, helping to determine whether a given robotic system can successfully execute the intended operations. By incrementally testing trajectories, optimizing grasping techniques, and analyzing execution efficiency, we gain a deeper understanding of how far COMPAS FAB can be pushed in the context of robotic fabrication.

Ultimately, the insights gained from this semi-interactive approach contribute not only to process optimization but also to the broader evaluation of COMPAS FAB as a tool for robotic motion planning and digital fabrication workflows.

4.4.1 Playground Overview for Assembly Tasks

The proposed workflow integrates robot initialization, collision object definition, motion planning, and trajectory execution to enable a structured approach to robotic motion simulation. By leveraging parametric design tools in Grasshopper, robotic trajectories can be defined for complex assembly processes. The following details the step-by-step implementation of this framework, highlighting its capabilities for robotic path planning and environment interaction.

The first step, as shown in figure (4.15), involves establishing a connection with the ROS client, which facilitates communication between Grasshopper and the robotic system. Once the connection is established, the robot model is loaded from a custom URDF file, incorporating the attached gripper to enable object manipulation. This initialization step ensures that the robotic system is accurately represented in the simulation environment.

To enable motion planning, the workspace must be defined with appropriate collision objects. A collision mesh representing a table is introduced to the environment, serving as a reference surface for the pick-and-place task. Additionally, objects can be integrated into the scene, including components placed at the initial position, the target location, or directly attached to the robotic gripper. These objects allow for the simulation of grasping and placing interactions within a constrained environment.

The object positions of the start and target planes are converted to frames to be compatible with the Compas fab frameworks motion planning functions. They guide the robotic arm's end-effector. These target locations are further categorized based on orientation, including horizontal, diagonal, and vertical components, depending on the required placement strategy.

The Start Configuration is the starting position, in which the robot is configured at the very beginning.

Once the robotic model and its environment are defined, the motion planning process is initiated. The Motion Plan component facilitates the computation of collision-free trajectories by utilizing the underlying COMPAS FAB motion planning pipeline. Two primary types of motion planning are employed: General Motion Planning determines an optimized path for the robot to transition from the start configuration to the target location while avoiding obstacles. While Cartesian Motion Planning ensures smooth, linear trajectories between targets, maintaining precise control over end-effector positioning. These planned trajectories define the movement strategy for the robot to execute the pick-and-place sequence accurately. The Motion Plan Componentś output is a list of configurations for every robot position required to move from the start to the target plane.

The final stage involves the visualization of the computed trajectories. The Visualization component provides a graphical representation of the planned motion, allowing for assessment and refinement before execution. Once validated, the motion plan is executed within the simulated environment, ensuring accurate replication of the pick-and-place sequence.

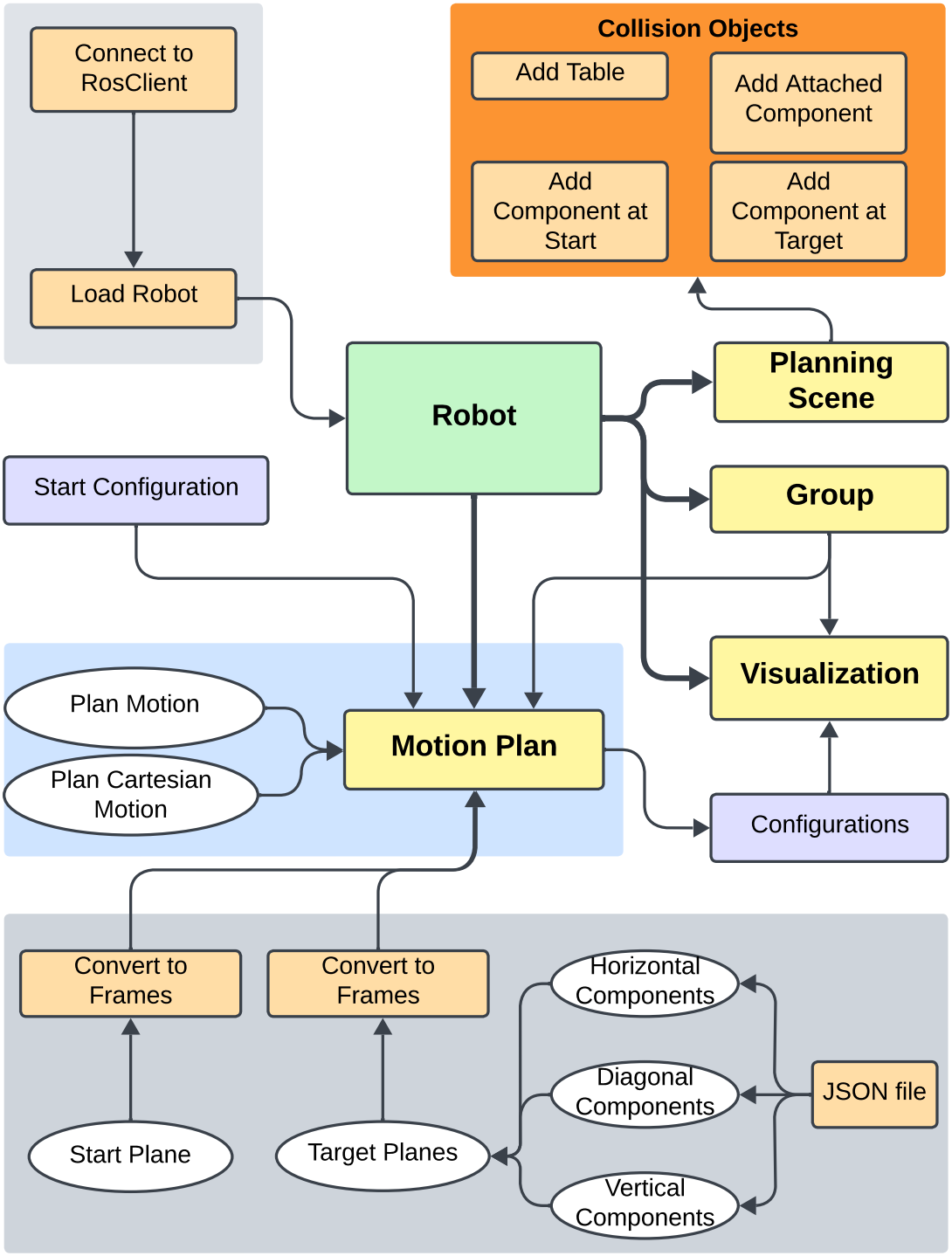


Figure 4.15: Grasshopper Playground Concept

4.4.2 Robot Workflow for a Single Pick-and-Place Cycle

Robotic pick-and-place operations are fundamental in automated assembly, requiring precise motion planning to ensure efficient and collision-free execution.

To illustrate the functionality of the Grasshopper Playground for assembly tasks, this subsection details the step-by-step execution of a single pick-and-place operation. The

workflow is structured to reflect the sequential actions the robot performs, from initialization to placing the component at its target location.

The process begins with loading the robot model into the scene and establishing a connection to the ROS client. The motion trajectory for the pick-and-place operation is then computed using the motion planning functionalities of COMPAS FAB. Once the trajectory is successfully generated, the robot moves into its start configuration, ensuring it begins from a predefined and repeatable initial position.

To accurately simulate the interaction between the robot and its environment, the table, where the robot is mounted on, is first added to the scene as a collision object. Following this, the first timber component, which will be manipulated by the robot, is introduced into the simulation as an additional collision mesh. These elements define the spatial constraints within which the robot must operate, allowing for precise collision-aware motion planning.

With the environment set up, the picking trajectory is initiated. The robot moves from its start configuration toward the predefined grasping position, ensuring optimal alignment with the component. Once the robot reaches the grasping position, the timber component is attached to the robot's gripper as a attached collision mesh. This step updates the robot's collision model, allowing the subsequent motion planning to account for the object being carried.

After successfully attaching the timber component, the placing trajectory is executed. The robot follows the computed motion path to transport the component from its initial position to the designated target location. Upon reaching the placing position, the component is added to the scene at the target plane, effectively detaching it from the robot's gripper. This transition completes the pick-and-place cycle, ensuring that the component is placed precisely while maintaining collision integrity within the environment.

This structured workflow provides an interactive yet systematic approach to robotic pick-and-place operations.

4.4.3 Grasshopper Component Blocks for Simulated Robotic Assembly

The image A.3 depicts the computational workflow developed in Grasshopper for Rhino, utilizing the COMPAS FAB framework to facilitate robotic fabrication and automation. The structured node-based setup enables seamless integration between digital design and robotic control. The ROS and Robot section establishes communication with the Robot Operating System (ROS), allowing users to load and control the robotic model. The Planning Scene and Group modules define the workspace and primary motion planning parameters, while the Attach Component and Add Components sections manage object manipulation, including attaching workpieces to the robot's end-effector. The Compute Paths module handles motion planning, enabling precise control of robotic movement and gripper operations. Additionally, the Component to Orientation node ensures proper alignment of components before execution, while the Table section provides an interactive

environment for adding or removing elements within the scene. The Visualization module enhances real-time monitoring by displaying the robot, collision objects, and reference frames. This Grasshopper-based workflow streamlines robotic assembly, enabling precise, automated execution and bridging the gap between computational design and physical fabrication.

Connecting to ROS Client, Loading the Robot and Setting up the Planning Scene

To connect to ROS and load the robot, the prebuilt COMPAS FAB Grasshopper components can be used. The ROS Connect Component links to the ROS Robot component, which is activated by toggling the load button.

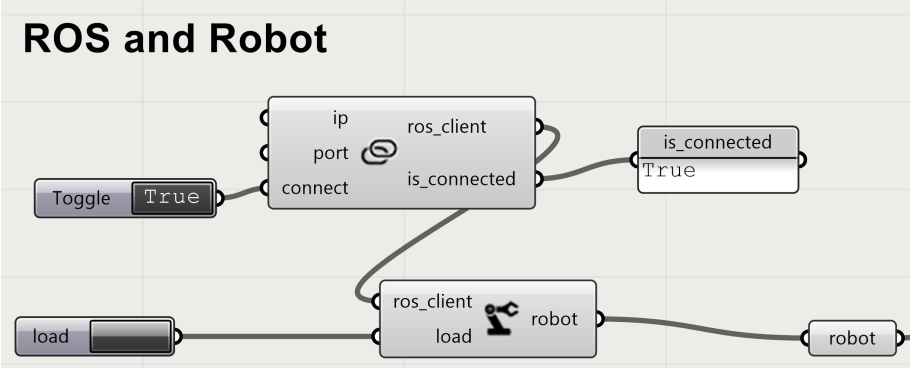


Figure 4.16: Connecting to ROS Client and Loading the Robot

To integrate the table onto which the robot arm is mounted, the prebuilt COMPAS FAB component Collision Mesh can be employed with a slight modification. Specifically, this line is to be deleted:

```
mesh = RhinoMesh.from_geometry(M).to_compas()
```

In its place the following line should be copied:

```
mesh = Mesh.from_stl(compas_fab.get('planning_scene/table.stl'))
```

In this case, the `Mesh.from_stl()` method is used to directly load a 3D mesh from an STL file located at planning_scene/table.stl. `The compas_fab.get()` function is responsible for retrieving the specified STL file from the given path. This method imports the mesh data in the STL format and converts it into a COMPAS-compatible mesh format, represented by the Mesh object in COMPAS. This approach is suitable for directly incorporating pre-existing geometric models, such as the floor or table

The Planning Scene component in Grasshopper can then be used to link the robot to the scene. It creates a virtual model of the robot's environment, encompassing objects, obstacles, and other relevant elements that interact with the robot during both planning and execution (RUST et al., 2018).

To extract the robot planning group, the COMPAS FAB function `info()` is used and implemented in a python component. The group is needed for motion planning as well as visualizing the simulation.

```
if robot:
    robot.info()
    main_group_name = robot.main_group_name
```

Importing Plane Data from JSON and Structuring in Orientation

This component processes and categorizes three-dimensional frames extracted from a JSON dataset using `compas_fab`, `compas.geometry`, and Rhino's geometry libraries. The JSON data, containing frame properties, is parsed into a structured dictionary, from which key geometric attributes such as position and axis vectors are extracted. For each frame, the script constructs Rhino geometry objects, including `Point3d` for position and `Vector3d` for orientation. A classification mechanism is implemented based on axis alignments: frames with a nonzero Z component in their Y-axis are identified as diagonal, those with a greater sum of the X component of the X-axis and the Z component of the Z-axis compared to the Z-axis and Y-axis components are categorized as horizontal, and frames where the X-axis component equals the Y-axis component are classified as vertical. Corresponding `rg.Plane` objects are generated, converted into `compas.geometry.Frame` instances, and stored in respective dictionaries. The categorized frames are then aggregated into a structured list, facilitating their use in robotic planning or computational design applications within Rhino and Grasshopper environments.

```
import compas_fab
from compas.geometry import Frame
import rhinoscriptsyntax as rs
import Rhino.Geometry as rg

points_json = compas_fab.get(json)
data = compas.json_load(points_json)

frame_json = data["properties"]
frames_h = {}
frames_v = {}
frames_d = {}

for nr, i in enumerate(frame_json):
    point_i = rg.Point3d(i["point"][0], i["point"][1], i["point"][2])
    x_axis_i =  rg.Vector3d(i["xaxis"][0], i["xaxis"][1], i["xaxis"][2])
    y_axis_i =  rg.Vector3d(i["yaxis"][0], i["yaxis"][1], i["yaxis"][2])
    z_axis_i =  rg.Vector3d(i["zaxis"][0], i["zaxis"][1], i["zaxis"][2])

    if i["yaxis"][2] != 0.0:
        pln_d = rg.Plane(point_i, y_axis_i)
        frame_diagonal = Frame(pln_d[0], pln_d[1], pln_d[2])
        frames_d[nr] = frame_diagonal
    elif i["xaxis"][0]+i["zaxis"][2] > i["zaxis"][2] + i["yaxis"][1]:
```

```
        pln_h = rg.Plane(point_i, -y_axis_i, -x_axis_i)
        frame_horizontal = Frame(pln_h[0], pln_h[1], pln_h[2])
        frames_h[nr] = frame_horizontal
    elif i["xaxis"][0] == i["yaxis"][1]:
        pln_v = rg.Plane(point_i, -z_axis_i, -y_axis_i)
        frame_vertical = Frame(pln_v[0], pln_v[1], pln_v[2])
        frames_v[nr] = frame_vertical

frames_lst = [frames_h, frames_v, frames_d]
```

Connecting Mesh Geometry of Timber Component to Frame and Orientation of Current Assembly Step

This component establishes the connection between the mesh geometry of a timber component and its corresponding frame and orientation within the current assembly step. Utilizing `compas_fab` and `compas.geometry`, it categorizes frames into three orientations—horizontal, vertical, and diagonal—based on a predefined frame list. A base `mesh_frame` is initialized at the world XY plane and slightly elevated in the Z direction. The script iterates through the categorized frames, identifying the current component's frame based on its assigned number. Depending on the orientation, the target frame is either directly assigned or transformed using a translation vector to align properly with the assembly structure. Mesh geometry for each orientation is loaded from STL files via `compas_fab.get()`, ensuring that the correct timber component model is associated with its designated frame. This structured approach enables precise placement and alignment of timber elements within a computational design and robotic fabrication workflow.

```
import compas_fab
from compas.datastructures import Mesh
from compas.geometry import Translation
from compas.geometry import Frame
import rhinoscriptsyntax as rs

frames_dict = {'h':frames_lst[0], 'v':frames_lst[1], 'd':frames_lst[2]}
mesh_frame = Frame.worldXY()
mesh_frame.point.z += 0.04/2

for orientation, frames in frames_dict.items():
    if orientation == 'h':
        for nr, frame in frames.items():
            if component_nr == nr:
                element = Mesh.from_stl(compas_fab.get('planning_scene/H_40.stl'))
                target_frame_cm = frames_dict[orientation][nr]
                target_frame_cm = target_frame_cm.transformed(Translation.from_vector([0,
    ↪ 0.002, 0]))
                target_frame = target_frame_cm.transformed(Translation.from_vector([0, 0,
    ↪ 0.02]))

    elif orientation == 'v':
        for nr, frame in frames.items():
```

```
            if component_nr == nr:
                target_frame_cm = frames_dict[orientation][nr]
                target_frame = target_frame_cm.transformed(Translation.from_vector([0.02,
    ↪ 0, 0]))
                element = Mesh.from_stl(compas_fab.get('planning_scene/V_40.stl'))

    elif orientation == 'd':
        for nr, frame in frames.items():
            if component_nr == nr:
                target_frame_cm = frames_dict[orientation][nr]
                target_frame = target_frame_cm.transformed(Translation.from_vector([0.0,
    ↪ -0.0164, 0.0115]))
                element = Mesh.from_stl(compas_fab.get('planning_scene/D_40.stl'))
```

Computing Start and Target Frames

To ensure the robot consistently picks up and places the component at a perpendicular angle, frames have to be added and transformed. Initially, the component placing frame is defined using a set of three vectors, representing the coordinate system's origin, x-axis, and z-axis, respectively. This frame is then subjected to a translation transformation, which is applied to define the approaching frame by translating the `place_t0cf_frame` further along the Z-axis by an `approach_offset`. Finally, the two transformed frames are stored in the list `start_frames`, which represents the different positions for the component handling.

```
import math
from compas.geometry import Frame
from compas.geometry import Transformation
from compas.geometry import Translation

place_t0cf_frame = Frame(place_t0cf_frame[0], place_t0cf_frame[1], place_t0cf_frame[2])
approach_t0cf_frame = place_t0cf_frame.transformed(Translation.from_vector([0, 0,
    ↪ approach_offset]))

start_frames = [approach_t0cf_frame, place_t0cf_frame]
```

The `pick_t0cf_frame` and `place_t0cf_frame` frames are created using specified coordinates and orientation vectors. These frames are then translated along the z-axis by an `approach_offset` value, resulting in the modified `pick_t0cf_frame` and a new `approach_t0cf_frame`. Finally, these frames are collected into a list called `target_frames`, which can be utilized for further robotic path planning or manipulation tasks. This method ensures precise control over the positioning and orientation of robotic end-effectors, enabling accurate and efficient task execution in automated systems.

```
from compas.geometry import Frame
from compas.geometry import Translation

pick_t0cf_frame = Frame(pick_t0cf_frame[0], pick_t0cf_frame[1], pick_t0cf_frame[2])
pick_t0cf_frame = pick_t0cf_frame.transformed(Translation.from_vector([0, 0,
    ↪ approach_offset]))
```

```
6
7  place_t0cf_frame = Frame(place_t0cf_frame[0], place_t0cf_frame[1], place_t0cf_frame[2])
8  approach_t0cf_frame = place_t0cf_frame.transformed(Translation.from_vector([0, 0,
       ↪ approach_offset]))
9
10 target_frames = [pick_t0cf_frame, approach_t0cf_frame, place_t0cf_frame]
```

Image 4.17 illustrates the relationship between planes and frames, as well as the connection of components to frame points and their corresponding orientations, as discussed above.

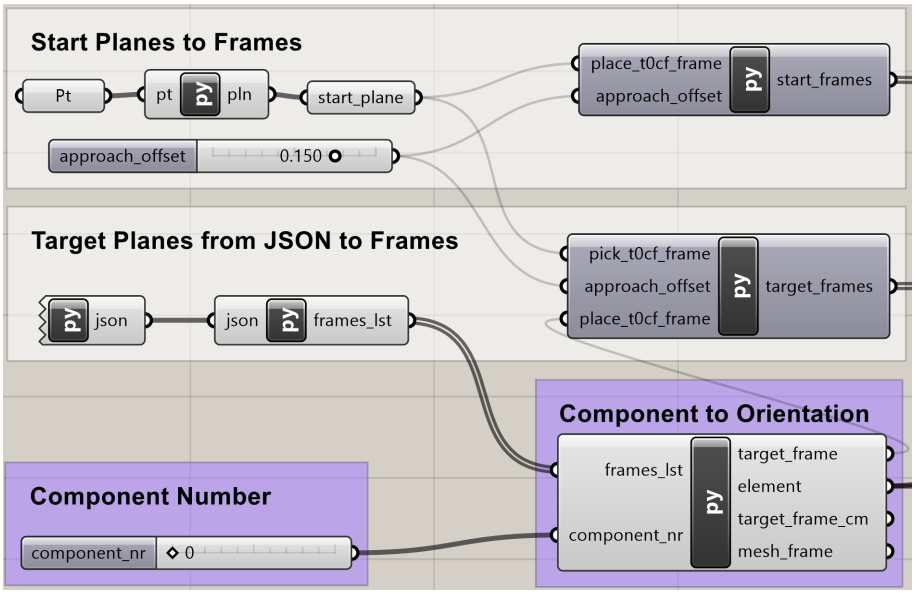


Figure 4.17: Importing Frame Data and Connecting Component

Motion Plan, Trajectory Computing and Gripper Configurations

The `PlanCustomMotion` Grasshopper Python component, A.4, computes a motion plan and trajectory for a robotic manipulator using COMPAS and COMPAS FAB. The component generates pick, place, and return trajectories based on user-defined input parameters, employing both Cartesian and free motion planning approaches.

The component is structured as a subclass of executingcomponent, ensuring its compatibility with the Grasshopper computational environment. The RunScript function defines the execution logic, receiving various input parameters such as the robot model, start and target frames, and configuration parameters:

```
1  class PlanCustomMotion(component):
2      def RunScript(
3          self, robot, attached_collision_mesh, start_planes, target_frames,
       ↪ start_configuration, open_gripper, close_gripper
4          , group, max_step, compute
5      ):
```

A unique identifier `pick_key` is generated to store the computed trajectory. The motion is planned in Cartesian space using the `plan_cartesian_motion` function. The start plane is converted into a COMPAS frame using `RhinoPlane.from_geometry`. If the robot is connected and computation is enabled, the motion is executed with constraints and optional collision checks:

```
pick_key = create_id(self, "pick_trajectory")
pick_path_constraints =  None
if robot and robot.client and robot.client.is_connected and compute:
    frames = [RhinoPlane.from_geometry(plane).to_compas_frame() for plane in [start_planes
    ↪ [1]]]
    st[pick_key] = robot.plan_cartesian_motion(
                        frames,
                        start_configuration=start_configuration,
                        group=group,
                        options=dict(
                            max_step=max_step,
                            path_constraints=pick_path_constraints,
                            attached_collision_meshes=attached_collision_mesh,
                        ),
                    )
pick_trajectory = st.get(pick_key, None)
```

The final configuration of the pick trajectory is used to close the robotic gripper. The closing motion generates a series of intermediate configurations leading to a fully closed state:

```
config = pick_configurations[-1]
close_gripper_configurations = close_gripper(config)
close_gripper_configuration = close_gripper_configurations[-1]
```

The placement trajectory follows a similar approach to the pick trajectory. The target frame for placement is converted into a COMPAS frame, and the `plan_cartesian_motion` function generates the path:

```

place_key = create_id(self, "place_trajectory")
place_path_constraints = None
if robot and robot.client and robot.client.is_connected and compute:
    frames = [RhinoPlane.from_geometry(plane).to_compas_frame() for plane in pplane]
    st[place_key] = robot.plan_cartesian_motion(
        frames,
        start_configuration=close_gripper_configuration,
        group=group,
        options=dict(
            max_step=max_step,
            path_constraints=place_path_constraints
        ),
    )
place_trajectory = st.get(place_key, None)
```

A second placement trajectory is computed using the `plan_motion` function, allowing for non-Cartesian motion. A `goal_constraint` is derived from the target frame with predefined tolerances, and the RRTConnect planner is used to find a feasible trajectory:

```
place2_key = create_id(self, "place2_trajectory")
planner_id = "RRTConnect"
place2_start_configuration = place_configurations[-1]
if robot and robot.client and robot.client.is_connected and compute:
    st[place2_key] = robot.plan_motion(
        goal_constraints,
        start_configuration=place2_start_configuration,
        group=group,
        options=dict(
            path_constraints=goal_constraints,
            planner_id=planner_id,
        ),
    )
place2_trajectory = st.get(place2_key, None)
```

After placement, the gripper opens by reversing the previous closing motion. The robot then computes a return trajectory to its initial position, made of a free motion planned trajectory and a cartesian motion planned trajectory:

```

open_config = place3_configurations[-1]
open_gripper_configurations = open_gripper(open_config)
open_gripper_configuration = open_gripper_configurations[-1]

return_key = create_id(self, "return_trajectory")
return_start_configuration = return1_configurations[-1]
if robot and robot.client and robot.client.is_connected and compute:
    st[return_key] = robot.plan_motion(
        return_goal_constraints,
        start_configuration=return_start_configuration,
        group=group,
        options=dict(
            path_constraints=return_goal_constraints,
            planner_id="RRTConnect",
        ),
    )
return_trajectory = st.get(return_key, None)
```

The computed configurations for all trajectories are aggregated into three sets:

```
final_pick_configurations = pick_configurations + close_gripper_configurations
final_place_configurations = place_configurations + place2_configurations +
    ↪ place3_configurations + open_gripper_configurations
final_return_configuration = return1_configurations + return_configurations

all = [final_pick_configurations, final_place_configurations, final_return_configuration]

return all
```

The following component defines a function `open_gripper` to generate a series of robot configurations that gradually open the robot's gripper. The function takes the current robot configuration as input and calculates the required joint values for each configuration. The gripper's opening is controlled by the variable steps, which defines the increment in joint values for each iteration. The function iterates five times, adjusting the gripper joint values and keeping other joint values unchanged. Each iteration creates a new configuration, which is stored in the list configurations. Finally, the list is reversed before being returned, providing the gripper opening sequence in the correct order.

```
from compas.robots import Configuration

def open_gripper(robot_config):
    pick_tolerance = 0.005
    width = 0.05
    steps = 0.001
    dis = (width - 0.04 - pick_tolerance)/2
    configurations = []
    joint_values = []

    for j in range(5):
        joint_values_i = []
        for i,t in enumerate(robot_config.joint_types):
            if t == 2:
                gripper_value = steps*j
                joint_values_i.append(gripper_value)
            else:
                joint_values_i.append(robot_config.joint_values[i])
        gripper_configuration_i = Configuration(joint_values_i, robot_config.joint_types,
    ↪ robot_config.joint_names)
        joint_values.append(joint_values_i)
        configurations.append(gripper_configuration_i)

    return list(reversed(configurations))
```

A similar component is implemented to close tge gripper fingers.

```
from compas.robots import Configuration

# Gripper Configuration
def close_gripper(robot_config):
    pick_tolerance = 0.005
    width = 0.05
    steps = 0.001
    dis = (width - 0.04 - pick_tolerance)/2
    configurations = []
    joint_values = []

    for j in range(5):
        joint_values_i = []
        for i,t in enumerate(robot_config.joint_types):
            if t == 2:
                gripper_value = steps*j
                joint_values_i.append(gripper_value)
```

```
            else:
                joint_values_i.append(robot_config.joint_values[i])
        gripper_configuration_i = Configuration(joint_values_i, robot_config.joint_types,
    ↪ robot_config.joint_names)
        joint_values.append(joint_values_i)
        configurations.append(gripper_configuration_i)

    return configurations
```

These two components are connected to the PlanCustomMotion component, so that the two functions can be used during the configuration calculation.

The `start_configuration` parameter needed for the PlanCustomMotion component is implemented in a componenet and then connected, which consists of the follwoing:

```
from compas.robots import Configuration

revolute = [1.366, -1.244, 1.856, -2.193, -1.578, 2.927]
prismatic = [0.0]
joint_names = [
    "shoulder_pan_joint",
    "shoulder_lift_joint",
    "elbow_joint",
    "wrist_1_joint",
    "wrist_2_joint",
    "wrist_3_joint",
    "hand_e_finger_2_joint"
]
joint_types = [0, 0, 0, 0, 0, 0, 2]
joint_values = revolute + prismatic

config = Configuration(joint_values, joint_types, joint_names)
```

The presented GhPython component PlanCustomMotion effectively integrates COMPAS FAB functionalities to compute motion plans for an articulated robotic system. By utilizing Cartesian and free-motion planning strategies, the component ensures flexibility and precision in executing pick-and-place tasks within a Grasshopper environment. The structured approach allows for modular and reusable motion planning, facilitating further extensions and optimizations. For further understanding of its connections to other components please refer to the image 4.18 below.

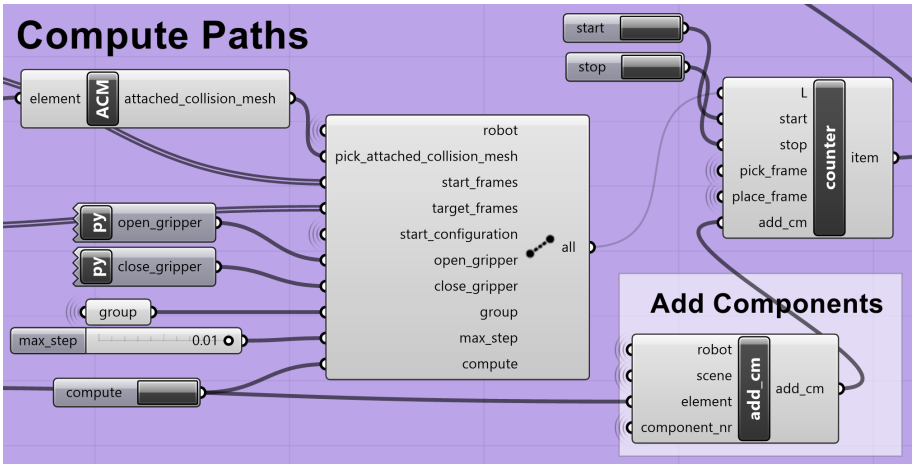


Figure 4.18: Motion Planning and Trajectory

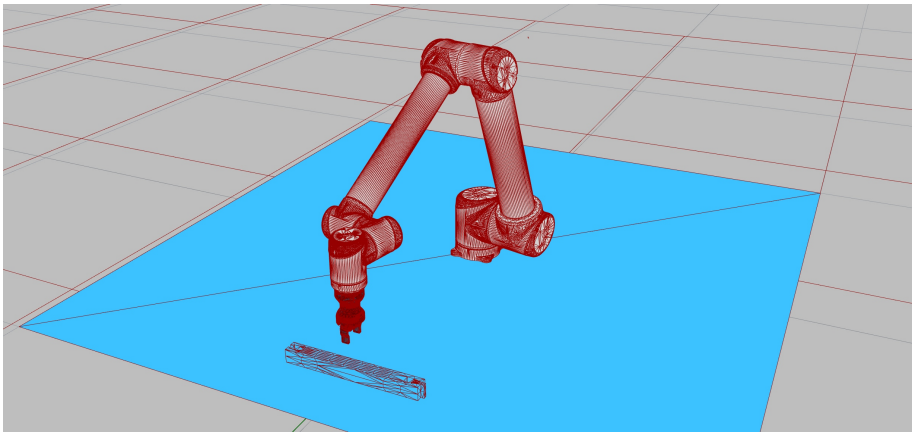


Figure 4.19: Robot Simulation accroding to Computed Motion Plan

After toggeling the compute button and then the start button the simulation in rhino (4.19) starts to assemble the component determined by the component number slider which can be seen in image 4.17.

Adding Collision Mesh Component

The function, `add_cm(frame)`, updates the collision environment in a robotic planning scene using COMPAS FAB. It starts by generating a unique identifier, `build_comp_id`, for the collision mesh based on `component_nr`. Any existing collision mesh with the same identifier is removed from the scene to prevent conflicts. The function then ensures that the input frame is properly formatted as a COMPAS Frame object, which defines a spatial reference for the collision mesh. A new `CollisionMesh` instance is created using the provided geometric element, unique identifier, and transformed frame. Before adding the new collision mesh, any previously attached collision meshes associated with the component are removed to avoid redundancy. The new collision mesh is then added to the planning scene, ensuring that it is considered during motion planning. Finally, `robot.client.get_planning_scene()` is called to synchronize the local planning scene

with the updated collision information, ensuring that the motion planner has an accurate representation of obstacles in the environment.

```
def add_cm(frame):
    build_comp_id = 'built_' + str(component_nr)
    scene.remove_collision_mesh(build_comp_id)
    frame = Frame(frame[0], frame[1], frame[2])
    cm = CollisionMesh(element, build_comp_id, frame)
    scene.remove_attached_collision_mesh('component_' + str(component_nr))
    scene.add_collision_mesh(cm)
    robot.client.get_planning_scene()
```

Adding Attached Collision Mesh Component

The component defines a function, `add_acm(config_jointvalues)`, which manages the addition of an Attached Collision Mesh to the robotic planning scene using COMPAS FAB. It begins by generating a unique identifier, `attached_id`, based on `component_nr`. If `config_jointvalues` is greater than zero, which is the finger joint value when the gripper is open, indicating that the component should be considered as attached to the robot, the function first removes any existing collision mesh with the corresponding `built_` identifier to prevent conflicts. A new `CollisionMesh` instance is then created using the given element, the generated `attached_id`, and `mesh_frame`, which provides the spatial reference. This collision mesh is subsequently wrapped into an `AttachedCollisionMesh` instance, specifying 'tcp' (tool center point) as the attachment reference, ensuring that the component moves with the robot's end effector. Finally, the attached collision mesh is added to the planning scene, integrating it into the motion planning framework to account for potential collisions during trajectory computation.

```
def add_acm(config_jointvalues):
    attached_id = 'component_' + str(component_nr)

    if config_jointvalues > 0:
        scene.remove_collision_mesh('built_' + str(component_nr))
        collision_mesh = CollisionMesh(element, attached_id, mesh_frame)
        attached_collision_mesh = AttachedCollisionMesh(collision_mesh, 'tcp')
        scene.add_attached_collision_mesh(attached_collision_mesh)
```

Counter for Configuration Iteration

The `runner(config_dict, start, stop, pick_frame, place_frame, add_cm)` function, A.5, manages the execution of a sequential motion process for a robotic system within a Grasshopper Python environment. It utilizes the scriptcontext.sticky dictionary to maintain persistent state variables across component executions.

First, it generates unique keys based on the component's unique identifier. These keys store the current motion step, iteration count, execution status, and last processed configuration.

The function follows a predefined motion sequence: Pick, Place and Return. If stop is True, the function halts execution and returns the last processed item. If start is True, the process begins or resumes execution.

The function determines the current motion step using motion_key and retrieves the corresponding configurations from `config_dict`. If the motion is "pick", it calls `add_cm(pick_frame)`to add a collision mesh for the pick location. If the motion is "return", it calls `add_cm(place_frame)` to add a collision mesh at the placement location.

The function then updates the counter to iterate through the configurations for the current motion. When a sequence is completed, it advances to the next motion type. Once all motion steps are executed, the function resets and stops execution.

Finally, `update_component(ghenv, 5)` ensures that the Grasshopper component is updated accordingly. The function returns the current configuration item, ensuring smooth and controlled motion transitions.

At the end, `config_dict_in` initializes the motion sequences using predefined lists `L[0]`, `L[1]`, `L[2]`, corresponding to pick, place, and return configurations. The runner function is then executed with these inputs to drive the motion process.

Visualizing the Robot Simulation

To visualize the robot, its movements, and all components, the COMPAS FAB component Robot Visualize is employed. The robot configurations are included through the configuration variable by connecting it to the item variable of the counter component. Minor adjustments are made to enable the use of the Attach Component Mesh function within the visualization component. The following is added to the visualiztion component after declaring the cached scene key and its identification.

```
cached_scene_key = create_id(self, "cached_scene")

# Add atached collision mesh
add_acm(configuration.joint_values[1])
```

The implementation of robotic fabrication using the COMPAS FAB framework has successfully demonstrated the feasibility of automating modular timber assembly. Through the development of a parametric joinery system, integration with robotic simulation tools, and execution of automated assembly workflows, this chapter has outlined key steps in bridging digital design with robotic execution. However, challenges remain, particularly in configuring robotic simulations, integrating adaptive end-effectors, and ensuring seamless interaction between COMPAS FAB and external tools like MoveIt! and ROS. Despite these hurdles, the results confirm the viability of robotic automation for timber construction, highlighting both the efficiency and adaptability of digital fabrication methods. Future work should focus on refining system configurations, improving documentation for robotic setup,

and expanding interoperability with broader automation workflows to enhance the adoption of robotic fabrication in the construction industry.

Chapter 5

Experimental Evaluation

In this chapter, the experimental evaluation of the implementation is presented, focusing on the performance of the robotic assembly process for a given timber structure in the developed Grasshopper playground. The objective is to assess how well the robot executes the task, examining key performance indicators such as velocity, acceleration, and precision during assembly.

5.1 Robotic Motion Behavior: Position, Velocity, and Acceleration

The generated plots, A.7, depict the robot's motion characteristics during a single picking, placing, and returning motion iteration in the assembly process, providing insights into its position, velocity, and acceleration over time. Each subplot represents a different aspect of the robot's movement, with multiple lines corresponding to individual joints, allowing for a comparative analysis of their respective behaviors. The position plot illustrates how the robot's joints change their spatial location over time, with smooth transitions indicating stable and controlled motion, while any abrupt changes or oscillations could suggest positioning inaccuracies or mechanical imprecisions. The velocity plot reflects the rate of change of position, where consistent curves suggest an optimized and predictable movement pattern, whereas sharp peaks or fluctuations might indicate jerky motion or suboptimal trajectory execution. Lastly, the acceleration plot represents the rate of change of velocity, with gradual acceleration and deceleration indicating smooth motion transitions, minimizing mechanical stress, while sharp spikes may point to excessive force application, which could lead to structural strain or reduced precision. Overall, the plots suggest a well-controlled robotic motion, as evidenced by the smooth position transitions, moderate velocity fluctuations, and balanced acceleration patterns. However, any irregularities in the curves might indicate the need for trajectory refinements, such as optimizing speed constraints or introducing additional smoothing techniques to enhance the robot's overall efficiency and accuracy in timber assembly.

5.2 Precision Issues in the Simulation

The images 5.1 and 5.2 illustrate a wireframe representation of the robotic assembly process, highlighting the geometric structure of the components and the robotic end-effector. One key precision issue observed in the simulation is the deviation between

the intended and actual positioning of components during assembly. These inaccuracies stem from multiple factors, including numerical errors in the simulation, discretization of collision detection algorithms, and floating-point arithmetic limitations. Additionally, as the complexity of the assembly increases, small positioning errors can accumulate, leading to misalignments in later stages. The image also demonstrates the interaction between the robot and the assembled structure, where minor inaccuracies in positioning can propagate through subsequent assembly steps. These deviations could require corrective actions such as re-adjustments or re-computations, increasing the overall simulation time.

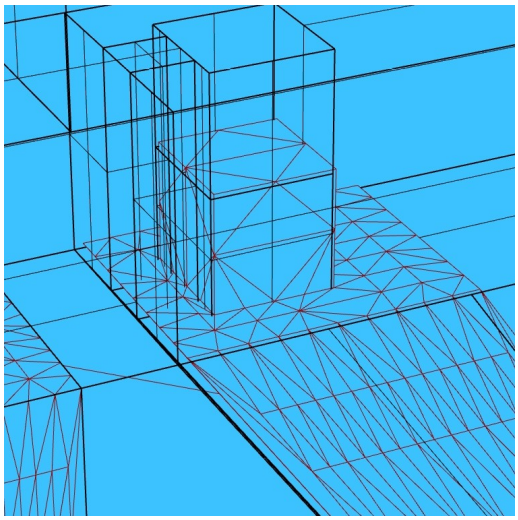


Figure 5.1: Inaccuracy Diagonal

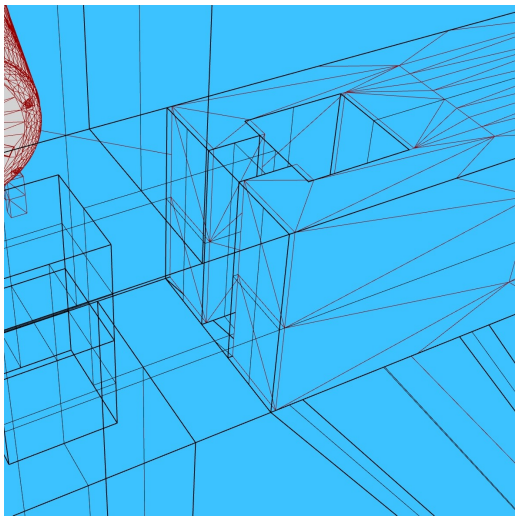


Figure 5.2: Inaccuracy Horizontal

5.3 Simulation Time and Scalability

The time required for assembling each component exhibits a significant increase throughout the process. While the first iteration of the first component takes approximately 30 seconds, the 11th component requires about 6 minutes to be placed correctly. This exponential growth in simulation time suggests a scaling issue, likely caused by increasing computational complexity as more objects are introduced into the environment. The image reveals a dense interaction between multiple assembled components, which may lead to an increased number of collision checks, constraint resolutions, and dynamic updates. Furthermore, as the assembly progresses, the system must account for a growing number of dependencies, which may lead to more frequent re-evaluations of the scene and prolonged computational effort. These factors highlight the importance of optimizing simulation algorithms, such as utilizing parallel processing, reducing precision where feasible, or implementing hierarchical collision detection to improve efficiency.

The following figures 5.3 depict the actual assembly process of the input structure data.

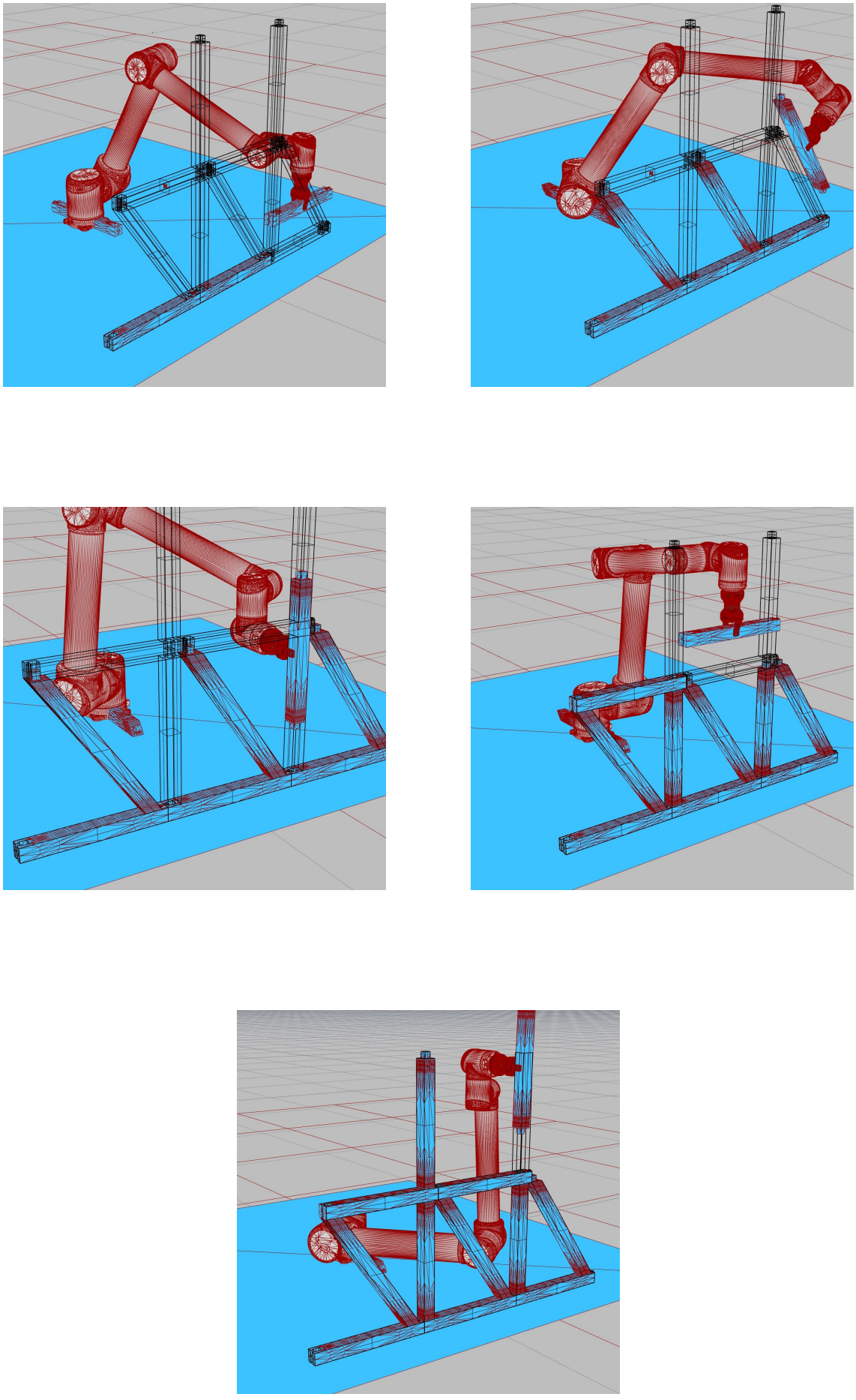


Figure 5.3: Real-Time Structure Assembly Simulation

5.4 Structural Analysis of Timber Structure

Finite element analysis (FEA) is a powerful numerical method used to simulate the behavior of structures under various loads. In the context of timber structures, FEA provides valuable insights into deformation, stress distribution, and reaction forces, which are essential for ensuring structural integrity and safety. For the purpose of calculation, the original timber structure has been simplified and broken down into its key components, to focus on the critical elements that influence its behavior. This simplification enables more efficient analysis while maintaining the essential characteristics of the structure.

The analysis involves a square frame with diagonal reinforcement, where the material properties and geometric characteristics of the beam are defined. Specifically, the elastic modulus, the cross-sectional area, and the moment of inertia are specified for the material. The nodal positions of the structure are defined for a simple square configuration with four nodes and the diagonal reinforcement, leading to five elements in total. The stiffness matrix for each element is computed using beam theory, which involves calculating the local stiffness matrix by considering the beam's elasticity, cross-sectional area, moment of inertia, and the angle between the nodes. These local stiffness matrices are assembled into a global stiffness matrix that represents the entire system. Boundary conditions are applied by fixing specific degrees of freedom (DOFs) corresponding to the nodes at positions 0, (x:0.0, y:0.0), and 1, (x: 0.4, y: 0.0), with the remaining DOFs being free. A vertical force of 5000 Newton is applied at Node 2. The displacement, 5.4, of the free nodes is computed by solving the reduced system of equations, and the reaction forces are determined by calculating the force residuals. The following input parameters were used:

$$E = 12 \times 10^9 \,\text{Pa} \tag{5.1}$$

$$A = 0.016 \,\text{m}^2 \tag{5.2}$$

$$I = 2.13 \times 10^{-7} \,\text{m}^4 \tag{5.3}$$

$$L = 0.4 \,\text{m} \tag{5.4}$$

$$F = -5000 \,\text{N} \tag{5.5}$$

The global stiffness matrix is assembled by summing the contributions of the local stiffness matrices for each element.

The load vector is:

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -5000 & 0 & 0 \end{bmatrix}^T$$

where a vertical force of 5000 N is applied at Node 2.

The reduced system of equations is:

$$\mathbf{K}_{\text{reduced}} \mathbf{U}_{\text{free}} = \mathbf{F}_{\text{reduced}}$$

The computed nodal displacements are:

$$\mathbf{U} = \begin{bmatrix} 0.00000000 & 0.00000000 \\ 0.00000000 & 0.00000000 \\ 1.04166667 \times 10^{-5} & -1.04166667 \times 10^{-5} \\ 1.04166667 \times 10^{-5} & 6.37836875 \times 10^{-22} \end{bmatrix}$$

The reaction forces are calculated as:

$$\mathbf{R} = \mathbf{K}_{\text{global}}\mathbf{U} - \mathbf{F}$$

The computed reaction forces are:

$$\mathbf{R} = \begin{bmatrix} -1.13686838 \times 10^{-12} & -1.13686838 \times 10^{-12} \\ 3.06161700 \times 10^{-13} & 5000 \\ 0.00000000 & 0.00000000 \\ 2.97171302 \times 10^{-13} & 5.04870979 \times 10^{-29} \end{bmatrix}$$

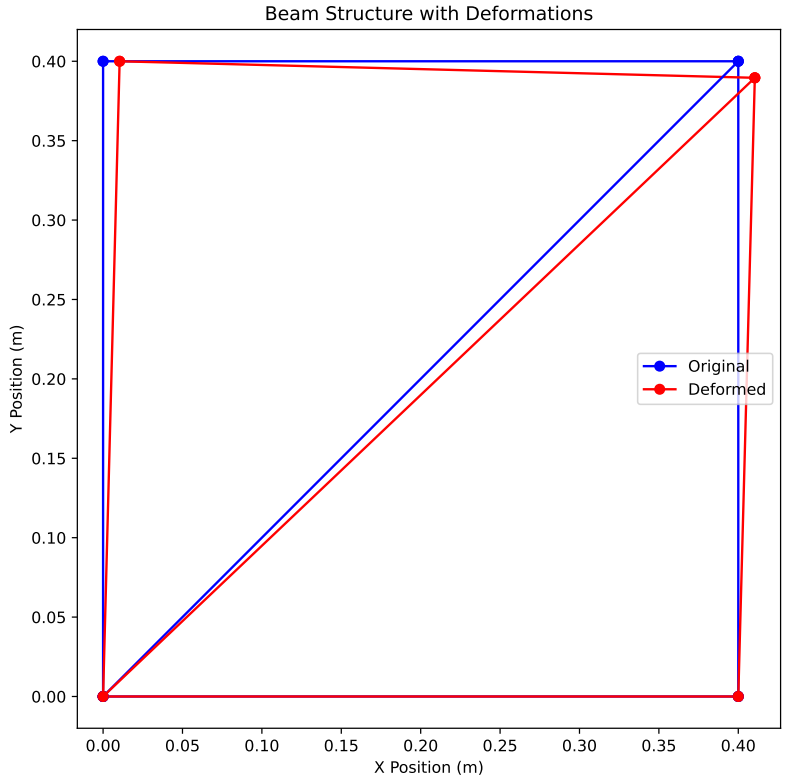


Figure 5.4: Deformations

The results of the structural analysis indicate that the displacements and reaction forces are very small, consistent with the high rigidity of the system. The nodal displacements are on the order of micrometers, with Nodes 0 and 1 (the fixed supports) showing no displacement, as expected. Node 2, where the vertical force is applied, exhibits a small displacement in both the x and y directions, while Node 3 experiences a similar displacement in the x direction but almost no displacement in the y direction, indicative of minimal vertical movement. The reaction forces at the fixed supports are very small, with Node 1 having a vertical reaction force of exactly 5000 N, which balances the applied load at Node 2. The reaction forces at Nodes 0 and 3 are negligibly small, likely due to numerical precision in the calculations, and do not significantly affect the overall equilibrium. These results confirm that the structure is behaving as expected under the applied load, with minimal deformation and reaction forces, which is typical for a highly rigid beam system.

Chapter 6

Conclusion and Future Work

This thesis has demonstrated the potential of robotic fabrication and automation in timber construction through the integration of the COMPAS FAB framework. The research highlights how robotic systems enhance precision, reduce labor dependency, and contribute to more sustainable building practices. Experimental evaluations confirm that automated workflows improve efficiency and scalability, particularly in modular timber construction (EVERSMANN et al., 2017).

Despite these advancements, challenges remain in the full-scale adoption of robotic construction, including technological integration, cost considerations, and the need for skilled professionals trained in robotics and digital fabrication. Future research should focus on refining robotic path planning algorithms, expanding interoperability with BIM systems, and exploring AI-driven automation to further optimize construction workflows.

The construction industry struggles to embrace innovative technologies, and the lag in innovation contributes to reduced productivity and efficiency (CAI and ZOU, 2022). However, timber construction, with its frequent use of prefabricated building components and highly digitalized design processes, is particularly well-suited for robotic fabrication (EVERSMANN et al., 2017). The key contributors to a more innovative and sustainable construction industry include enhanced BIM development and widespread adoption within the sector. These advancements will enable further technological integration, such as robotic fabrication, to flourish, ultimately transforming how buildings are designed and constructed.

Working with COMPAS FAB provided valuable insights into both the potential and the limitations of the framework. While its documentation is comprehensive in some aspects, certain critical processes, such as attaching a gripper to a robotic arm, were insufficiently explained. The lack of guidance on configuring moving components, such as gripper fingers, added significant complexity to the workflow. Additionally, setting up the software environment was challenging for those unfamiliar with ROS and Linux. On a Windows machine, configuring Docker and establishing a functioning development environment required additional effort, especially when generating custom URDF files and MoveIt! configurations. The process was not well-documented, particularly in regard to selecting appropriate joints and links when configuring the MoveIt! package. These challenges highlight areas for improvement in the COMPAS FAB documentation and suggest that future iterations should provide more detailed tutorials and troubleshooting guidance for users with varying levels of experience in robotic simulation.

Looking ahead, future work could focus on implementing the entire system in real-life settings with actual robotic hardware to test the framework's practical applicability and

robustness. This would involve integrating the simulated robotic systems with real-world timber components, optimizing the design-to-fabrication workflow, and refining robotic motion capabilities for improved precision and efficiency. Additionally, further investigation into the use of CNC milling for timber component production could be explored, as it would enhance the integration of digital fabrication and traditional woodworking methods, potentially reducing waste and improving overall production speed.

Another exciting avenue for future work involves exploring the potential of incorporating augmented reality (AR) or virtual reality (VR) to assist operators in real-time, offering intuitive interfaces for assembly tasks or facilitating better communication between design teams and robotic systems. Finally, scaling the robotic systems to handle larger construction projects or more complex geometries could provide valuable insights into the viability of robotic automation in larger-scale timber construction, paving the way for broader adoption across the construction industry.

Ultimately, the findings of this thesis underscore the transformative role of robotics in the construction industry. By leveraging digital design tools and automation, the sector can achieve greater productivity, enhanced safety, and reduced environmental impact, paving the way for a new era of innovation in timber construction.

Appendix A

Appendix 1

A.1 hand_e.xacro

```
<!-- Here we define the 2 parameters of the macro -->
<xacro:macro name="hand_e" params="prefix connected_to">

  <!-- Create a fixed joint with a parameterized name. -->
  <joint name="${prefix}hand_e_base_joint" type="fixed">

    <!-- The parent link must be read from the robot model it is attached to. -->
    <parent link="${connected_to}"/>
    <child link="${prefix}hand_e_base"/>

    <!-- The tool is directly attached to the flange. -->
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </joint>

  <link name="${prefix}hand_e_base">
    <visual>
      <geometry>
        <!-- The path to the visual meshes in the package. -->
        <mesh filename="package://ur10_hand_e/meshes/visual/hand_e_base.stl"/>
      </geometry>
    </visual>
    <collision>
      <geometry>
        <!-- The path to the collision meshes in the package. -->
        <mesh filename="package://ur10_hand_e/meshes/collision/hand_e_base.stl"/>
      </geometry>
    </collision>
  </link>

  <joint name="${prefix}hand_e_finger_1_joint" type="prismatic">
    <!-- The parent link must be read from the robot model it is attached to. -->
    <parent link="${prefix}hand_e_base"/>
    <child link="${prefix}hand_e_finger_1"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <axis xyz="0 1 0"/>
    <limit effort="130" lower="0" upper="0.025" velocity="0.15"/>
  </joint>

  <link name="${prefix}hand_e_finger_1">
    <visual>
      <geometry>
        <!-- The path to the visual meshes in the package. -->
```

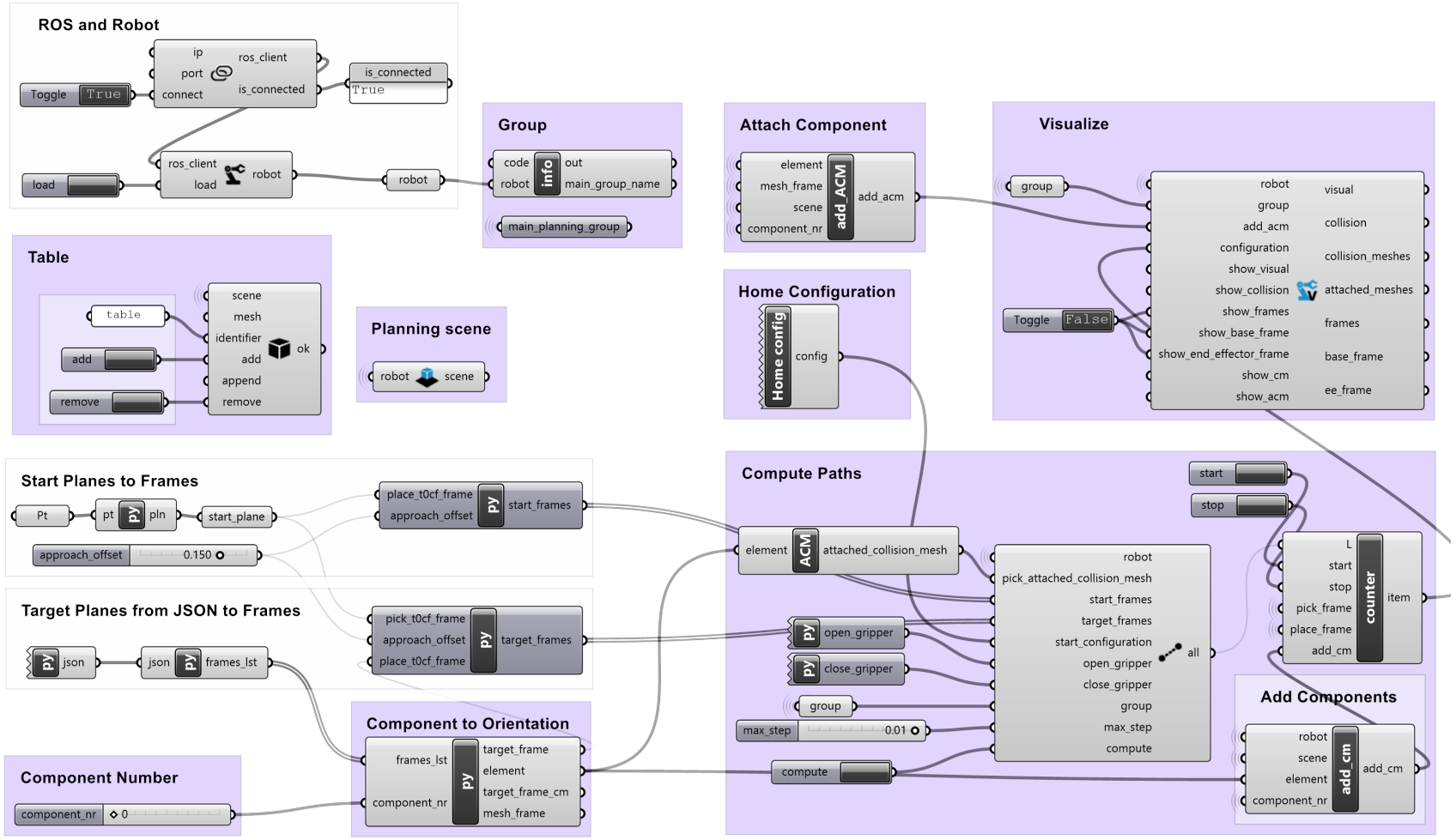
```
          <mesh filename="package://ur10_hand_e/meshes/visual/hand_e_finger_1.stl"/>
        </geometry>
      </visual>
      <collision>
        <geometry>
          <!-- The path to the collision meshes in the package. -->
          <mesh filename="package://ur10_hand_e/meshes/collision/hand_e_finger_1.stl"/>
        </geometry>
      </collision>
    </link>

    <joint name="${prefix}hand_e_finger_2_joint" type="prismatic">
      <!-- The parent link must be read from the robot model it is attached to. -->
      <parent link="${prefix}hand_e_base"/>
      <child link="${prefix}hand_e_finger_2"/>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <axis xyz="0 -1 0"/>
      <limit effort="130" lower="0" upper="0.025" velocity="0.15"/>
      <mimic joint="${prefix}hand_e_finger_1_joint" multiplier="1" offset="0"/>
    </joint>
    <link name="${prefix}hand_e_finger_2">
      <visual>
        <geometry>
          <!-- The path to the visual meshes in the package. -->
          <mesh filename="package://ur10_hand_e/meshes/visual/hand_e_finger_2.stl"/>
        </geometry>
      </visual>
      <collision>
        <geometry>
          <!-- The path to the collision meshes in the package. -->
          <mesh filename="package://ur10_hand_e/meshes/collision/hand_e_finger_2.stl"/>
        </geometry>
      </collision>
    </link>

    <!-- TCP frame -->
    <joint name="${prefix}tcp_joint" type="fixed">
      <origin xyz="0 0 0.116" rpy="0 0 0"/>
      <parent link="${prefix}hand_e"/>
      <child link="${prefix}tcp"/>
    </joint>
    <link name="${prefix}tcp"/>

  </xacro:macro>
</robot>
```

A.2 extract_coordinates_from_step.py

```python
import OCC.Core.STEPControl as step
import OCC.Core.BRepGProp as brep_gprop
import OCC.Core.GProp as gprop
import OCC.Core.TopoDS as topo_ds
import OCC.Core.TopExp as top_exp
import OCC.Core.TopAbs as top_abs
import OCC.Core.IFSelect as ifselect
import json

def extract_coordinates_from_step(filepath):
    '''
    Function for extracting center of mass and orientation
    of parametrized modular timber components from .step files
    and creating a json file to be imported into
    Grasshopper Playground for timber structure assembly tasks

    parameters: filepath
        filepath to the .step file
    return: output_data
        properties of timber components(center of mass and orientation)
    '''

    # Load the STEP file
    step_reader = step.STEPControl_Reader()
    status = step_reader.ReadFile(filepath)

    solids_with_centers = []

    if status == ifselect.IFSelect_RetDone:
        step_reader.TransferRoots()
        shape = step_reader.OneShape()
        i = 0
        coords = {}

        # Iterate through solids in STEP file
        exp = top_exp.TopExp_Explorer(shape, top_abs.TopAbs_SOLID)
        while exp.More():
            shape = exp.Current()

            solid = topo_ds.topods_Solid(shape)

            # Center of mass
            props = gprop.GProp_GProps()
            brep_gprop.brepgprop_VolumeProperties(solid, props)
            center_mass = props.CentreOfMass()

            center_x = round(center_mass.X() / 1000, 3)
            center_y = round(center_mass.Y() / 1000, 3)
            center_z = round(center_mass.Z() / 1000, 3)

            # Inertia matrix and axis calculation
```

```
            inertia_matrix = props.MatrixOfInertia()
            axis_x = (round(inertia_matrix.Value(1, 1)/ 1000000.0, 3),
                    round(inertia_matrix.Value(1, 2)/ 1000000.0, 3),
                    round(inertia_matrix.Value(1, 3)/ 1000000.0, 3))
            axis_y = (round(inertia_matrix.Value(2, 1)/ 1000000.0, 3),
                    round(inertia_matrix.Value(2, 2)/ 1000000.0, 3),
                    round(inertia_matrix.Value(2, 3)/ 1000000.0, 3))
            axis_z = (round(inertia_matrix.Value(3, 1)/ 1000000.0, 3),
                    round(inertia_matrix.Value(3, 2)/ 1000000.0, 3),
                    round(inertia_matrix.Value(3, 3)/ 1000000.0, 3))

            solids_with_centers.append({
                "point": [center_x, center_y, center_z],
                "xaxis": axis_x,
                "yaxis": axis_y,
                "zaxis": axis_z
            })

            exp.Next()

    # Sort the solids
    sorted_solids = sorted(solids_with_centers, key=lambda item: item["point"][2])

    output_data = {
        "properties": {},
        "required": ["point", "xaxis", "yaxis"]
    }

    output_data["properties"] = sorted_solids

    with open("points_axis.json", "w") as json_file:
        json.dump(output_data, json_file, indent=4)

    return output_data
```

A.3 Grasshopper Playground for Robotic Timber Assembly



A.4 class PlanCustomMotion(component)

```
from compas_rhino.conversions import RhinoPlane
from ghpythonlib.componentbase import executingcomponent as component
from scriptcontext import sticky as st
from compas_fab.ghpython.components import create_id
import rhinoscriptsyntax as rs
import math

class PlanCustomMotion(component):
    def RunScript(
        self, robot, attached_collision_mesh, start_planes, target_frames, start_configuration, open_gripper, close_gripper
        , group, max_step, compute
    ):
        pick_key = create_id(self, "pick_trajectory")
        pick_path_constraints =  None
        if robot and robot.client and robot.client.is_connected and compute:
            frames = [RhinoPlane.from_geometry(plane).to_compas_frame() for plane in [start_planes[1]]]
            st[pick_key] = robot.plan_cartesian_motion(
                                frames,
                                start_configuration=start_configuration,
                                group=group,
                                options=dict(
                                    max_step=max_step,
                                    path_constraints=pick_path_constraints,
                                    attached_collision_meshes=attached_collision_mesh,
                                ),
                            )
        pick_trajectory = st.get(pick_key, None)

        # Visualize Pick Trajectory
        pick_configurations = []
        if robot and pick_trajectory:
            group = group or robot.main_group_name
            for c in pick_trajectory.points:
                pick_configurations.append(
                    robot.merge_group_with_full_configuration(c, pick_trajectory.start_configuration, group)
                )
                pick_frame = robot.forward_kinematics(c, group, options=dict(solver="model"))

        pick_start_configuration = pick_trajectory.start_configuration

        # Gripper Configuration
        config = pick_configurations[-1]
        close_gripper_configurations = close_gripper(config)

        close_gripper_configuration = close_gripper_configurations[-1]

        # Place Trajectory Motion Plan Cartesian
```

```
        pplane = [target_frames[0]]
        place_key = create_id(self, "place_trajectory")
        place_path_constraints = None
        if robot and robot.client and robot.client.is_connected and compute:
            frames = [RhinoPlane.from_geometry(plane).to_compas_frame() for plane in
    ↪ pplane]
            st[place_key] = robot.plan_cartesian_motion(
                frames,
                start_configuration=close_gripper_configuration,
                group=group,
                options=dict(
                    max_step=max_step,
                    path_constraints=place_path_constraints
                ),
            )
        place_trajectory = st.get(place_key, None)

        # Visualize Place Trajectory
        place_start_configuration = None
        place_configurations = []
        if robot and place_trajectory:
            group = group or robot.main_group_name

            for c in place_trajectory.points:
                place_configurations.append(
                    robot.merge_group_with_full_configuration(c, place_trajectory.
    ↪ start_configuration, group)
                )
                place_frame = robot.forward_kinematics(c, group, options=dict(solver="
    ↪ model"))

            place_start_configuration = place_trajectory.start_configuration

        # Constraints
        goal_constraints = None
        if robot and target_frames:
            tolerance_position = 0.001
            tolerance_xaxis = 1.0
            tolerance_yaxis = 1.0
            tolerance_zaxis = 1.0

            p2place = target_frames[1]
            constraint_frame = RhinoPlane.from_geometry(p2place).to_compas_frame()
            tolerances_axes = [
                math.radians(tolerance_xaxis),
                math.radians(tolerance_yaxis),
                math.radians(tolerance_zaxis),
            ]
            goal_constraints = robot.constraints_from_frame(constraint_frame,
    ↪ tolerance_position, tolerances_axes, group)

        # Place 2 Trajectory Motion Plan Free
        place2_key = create_id(self, "place2_trajectory")
```

```
97          planner_id = "RRTConnect"
98          place2_start_configuration = place_configurations[-1]
99          if robot and robot.client and robot.client.is_connected and compute:
100             st[place2_key] = robot.plan_motion(
101                 goal_constraints,
102                 start_configuration=place2_start_configuration,
103                 group=group,
104                 options=dict(
105                     path_constraints=goal_constraints,
106                     planner_id=planner_id,
107                 ),
108             )
109         place2_trajectory = st.get(place2_key, None)
110
111         # Visualize Place 2 Trajectory
112         place2_start_configuration = None
113         place2_configurations = []
114
115         if robot and place2_trajectory:
116             group = group or robot.main_group_name
117
118             for c in place2_trajectory.points:
119                 place2_configurations.append(
120                     robot.merge_group_with_full_configuration(c, place2_trajectory.
    ↪ start_configuration, group)
121                 )
122                 place2_frame = robot.forward_kinematics(c, group, options=dict(solver="
    ↪ model"))
123
124             place2_start_configuration = place2_trajectory.start_configuration
125
126         # Place 3 Trajectory Motion Plan cartesian
127         pp3plane = [target_frames[2]]
128
129         place3_key = create_id(self, "place3_trajectory")
130         place3_start_configuration = place2_configurations[-1]
131         place3_path_constraints = None
132         if robot and robot.client and robot.client.is_connected and compute:
133             frames = [RhinoPlane.from_geometry(plane).to_compas_frame() for plane in
    ↪ pp3plane]
134             st[place3_key] = robot.plan_cartesian_motion(
135                 frames,
136                 start_configuration=place3_start_configuration,
137                 group=group,
138                 options=dict(
139                     max_step=max_step,
140                     path_constraints=place3_path_constraints
141                 ),
142             )
143
144         place3_trajectory = st.get(place3_key, None)
145
146         # Visualize Place 3 Trajectory
```

```
147         place3_start_configuration = None
148         place3_configurations = []
149 
150         if robot and place3_trajectory:
151             group = group or robot.main_group_name
152 
153             for c in place3_trajectory.points:
154                 place3_configurations.append(
155                     robot.merge_group_with_full_configuration(c, place3_trajectory.
    ↪ start_configuration, group)
156                 )
157                 place3_frame = robot.forward_kinematics(c, group, options=dict(solver="
    ↪ model"))
158 
159             place3_start_configuration = place3_trajectory.start_configuration
160 
161         # Open Gripper Configuration
162         open_config = place3_configurations[-1]
163         open_gripper_configurations = open_gripper(open_config)
164         open_gripper_configuration = open_gripper_configurations[-1]
165 
166         # Return 1 Trajectory Motion Plan cartesian
167         return1_key = create_id(self, "return1_trajectory")
168         return1_start_configuration = open_gripper_configuration
169         if robot and robot.client and robot.client.is_connected and compute:
170             st[return1_key] = robot.plan_cartesian_motion(
171                 [target_frames[1]],
172                 start_configuration=return1_start_configuration,
173                 group=group,
174                 options=dict(
175                     max_step=max_step,
176                     )
177             )
178 
179         return1_trajectory = st.get(return1_key, None)
180 
181         # Visualize Return 1 Trajectory
182         return1_configurations = []
183 
184         if robot and return1_trajectory:
185             group = group or robot.main_group_name
186 
187             for c in return1_trajectory.points:
188                 return1_configurations.append(
189                     robot.merge_group_with_full_configuration(c, return1_trajectory.
    ↪ start_configuration, group)
190                 )
191                 return1_frame = robot.forward_kinematics(c, group, options=dict(solver="
    ↪ model"))
192 
193         # Return Constraints
194         return_tolerances_axes = [math.radians(1.0), math.radians(1.0), math.radians(1.0)]
```

```
195         return_goal_constraints = robot.constraints_from_frame(start_planes[0], 0.001,
    ↪ tolerances_axes, group)
196 
197         # Return 2 Motion Plan Free
198         return_key = create_id(self, "return_trajectory")
199         return_start_configuration = return1_configurations[-1]
200         if robot and robot.client and robot.client.is_connected and compute:
201             st[return_key] = robot.plan_motion(
202                 return_goal_constraints,
203                 start_configuration=return_start_configuration,
204                 group=group,
205                 options=dict(
206                     path_constraints=return_goal_constraints,
207                     planner_id="RRTConnect",
208                 ),
209             )
210         return_trajectory = st.get(return_key, None)
211 
212         # Visualize Return 2 Trajectory
213         return_configurations = []
214 
215         if robot and return_trajectory:
216             group = group or robot.main_group_name
217 
218             for c in return_trajectory.points:
219                 return_configurations.append(
220                     robot.merge_group_with_full_configuration(c, return_trajectory.
    ↪ start_configuration, group)
221                 )
222                 return_frame = robot.forward_kinematics(c, group, options=dict(solver="
    ↪ model"))
223 
224             return_start_configuration = return_trajectory.start_configuration
225 
226         final_pick_configurations = pick_configurations + close_gripper_configurations
227         final_place_configurations = place_configurations + place2_configurations +
    ↪ place3_configurations + open_gripper_configurations
228         final_return_configuration = return1_configurations + return_configurations
229 
230         all = [final_pick_configurations, final_place_configurations,
    ↪ final_return_configuration]
231 
232         return all
```

A.5 def runner(config_dict, start, stop, pick_frame, place_frame, add_cm)

```
from scriptcontext import sticky as st
from compas_ghpython.utilities import update_component

def runner(config_dict, start, stop, pick_frame, place_frame, add_cm):

    # Store data in component
    guid = str(ghenv.Component.InstanceGuid)
    motion_key = "motion_" + guid
    counter_key = "counter_" + guid
    running_key = "running_" + guid
    last_item_key = "last_item_" + guid

    motion_order = ['pick', 'place', 'return']
    if motion_key not in st:
        st[motion_key] = 0

    if counter_key not in st:
        st[counter_key] = 0
    if running_key not in st:
        st[running_key] = False
    if last_item_key not in st:
        st[last_item_key] = config_dict[motion_order[0]][0]

    # If stop is True, return last item and exit
    if stop:
        st[running_key] = False
        return st[last_item_key]

    if start:
        st[running_key] = True

    if not st[running_key]:
        return st[last_item_key]

    motion = motion_order[st[motion_key]]
    configs = config_dict[motion]

    # Perform motion-specific actions
    if motion == 'pick':
        add_cm(pick_frame)
    elif motion == 'return':
        add_cm(place_frame)

    if 0 <= st[counter_key] < len(configs):
        item = configs[st[counter_key]]
        st[last_item_key] = item

    # Update counter for next iteration
    if st[counter_key] < len(configs) - 1:
```

```
        st[counter_key] += 1
    else:
        st[counter_key] = 0
        st[motion_key] += 1
        if st[motion_key] >= len(motion_order):
            st[motion_key] = 0
            st[running_key] = False

    update_component(ghenv, 5)

    return item

config_dict_in = {'pick': L[0], 'place': L[1], 'return': L[2]}
item = runner(config_dict_in, start, stop, pick_frame, place_frame, add_cm)
```

A.6 2D Drawings of Timber Components

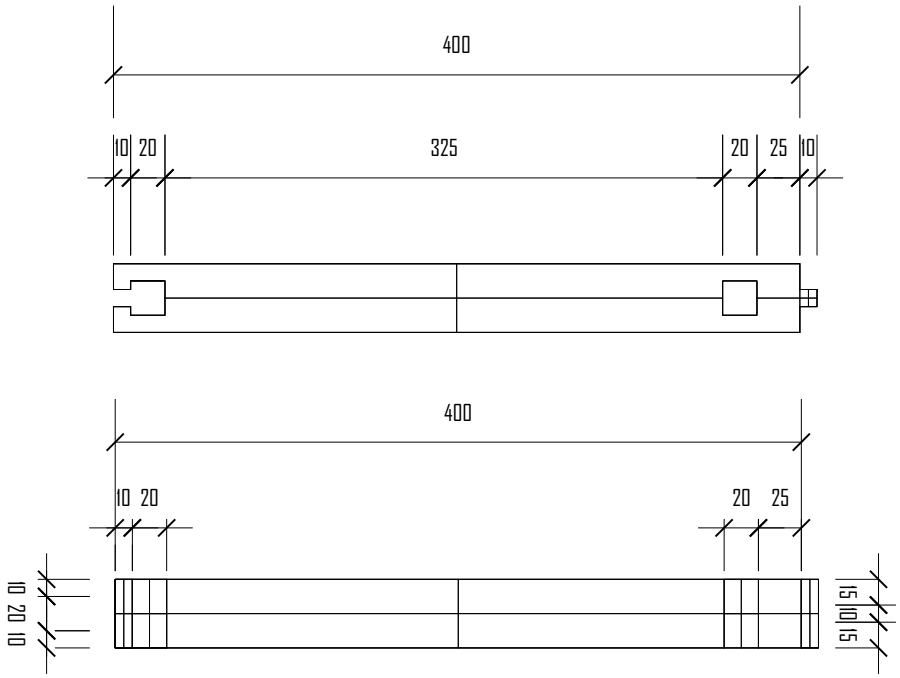


Figure A.1: Top and Side View of Horizontal Component in Millimeters

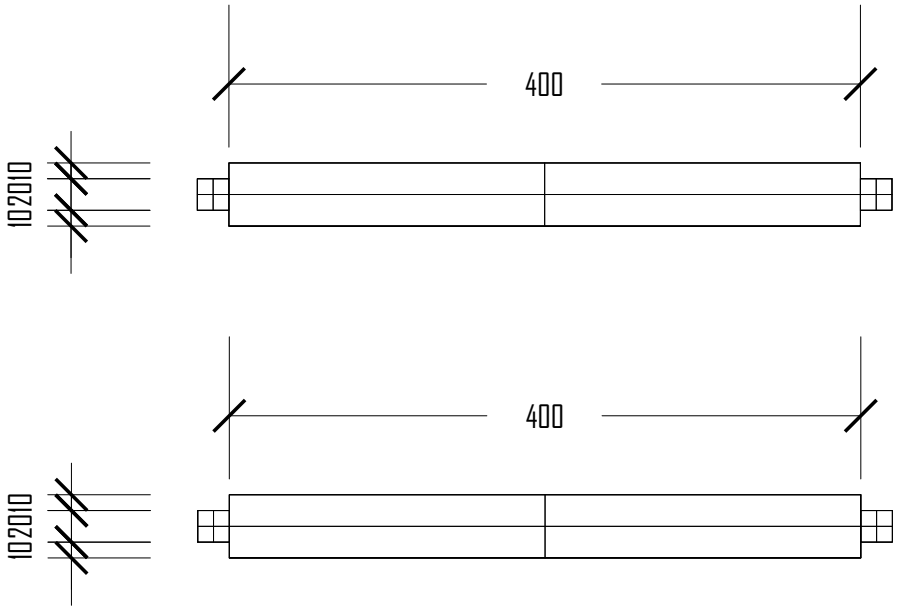


Figure A.2: Top and Side View of Vertical Component in Millimeters

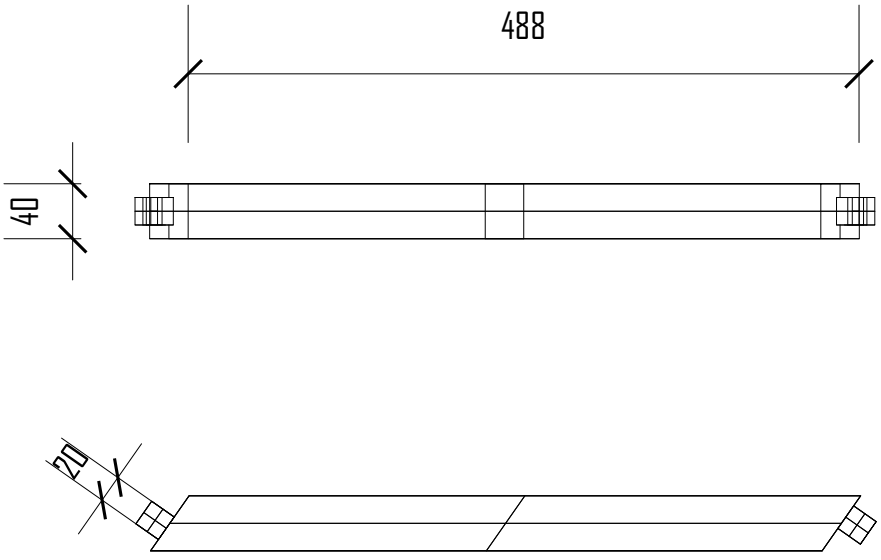


Figure A.3: Top and Side View of Diagonal Component in Millimeters

A.7 Acceleration, Velocity and Position Data for one Iteration

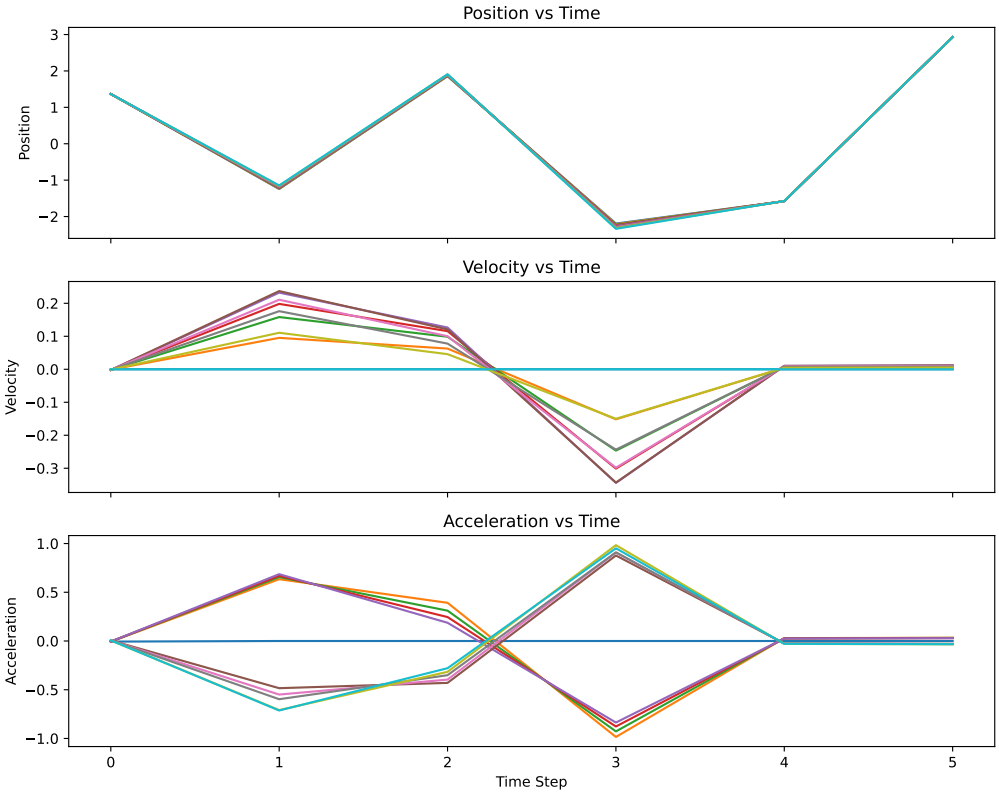


Figure A.4: Picking Motion Trajectory Data

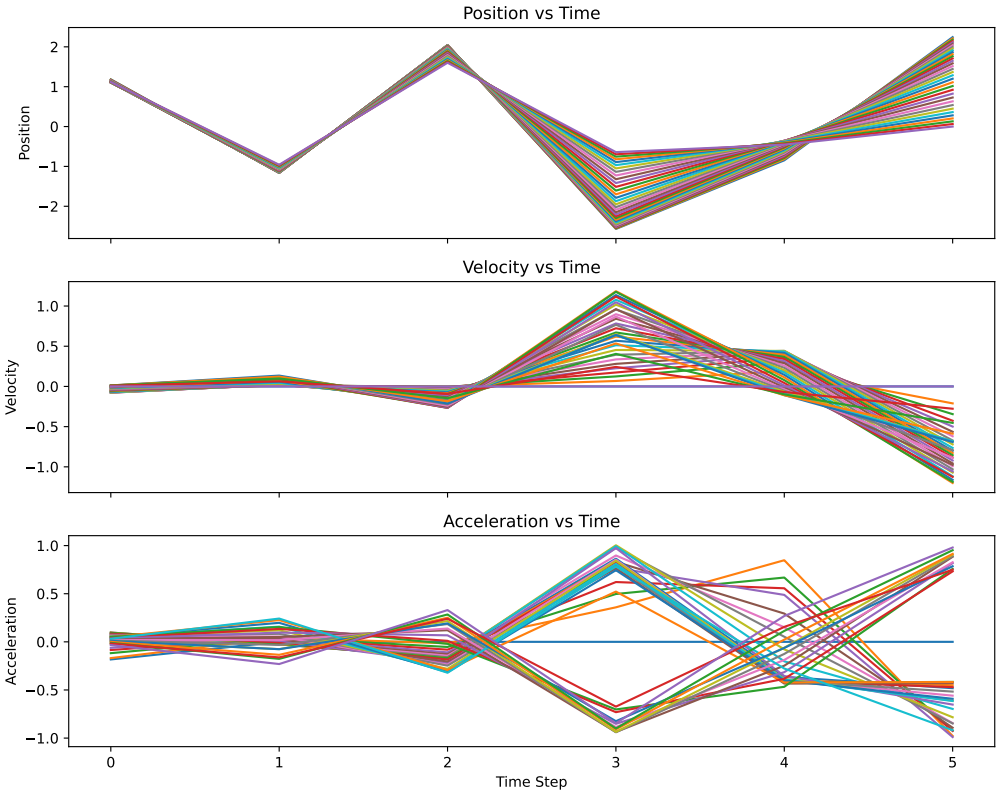


Figure A.5: Placing Cartesian Motion Trajectory Data

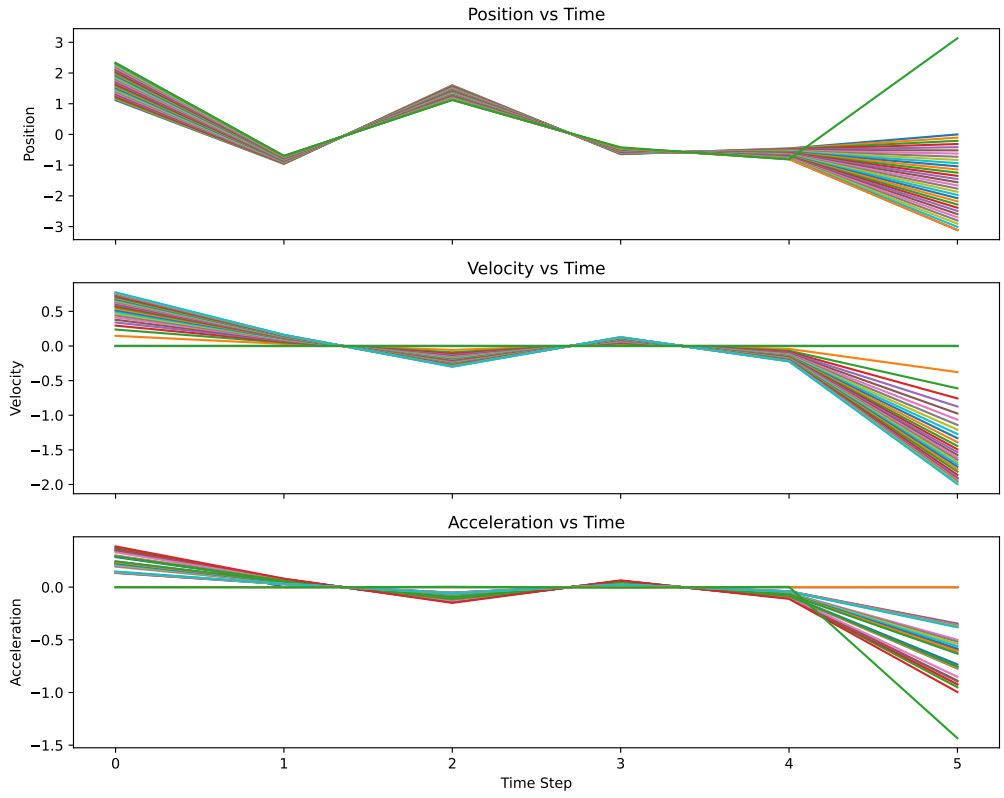


Figure A.6: Placing Free Motion Trajectory Data

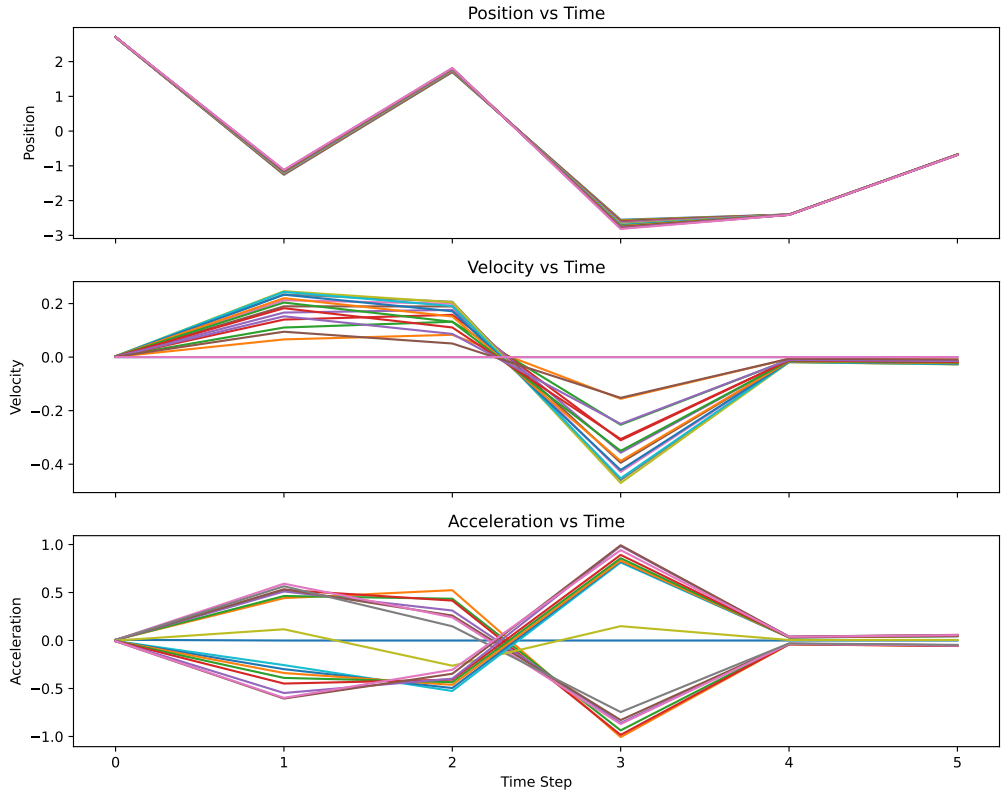


Figure A.7: Placing Target Cartesian Motion Trajectory Data

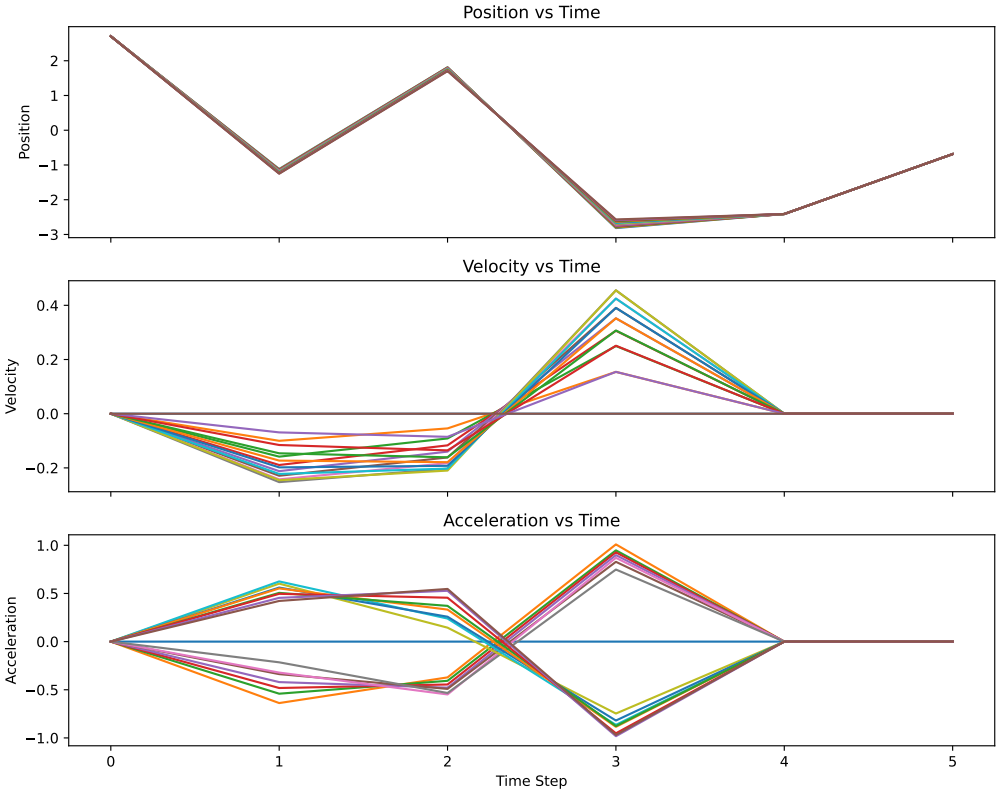


Figure A.8: Return Cartesian Motion Trajectory Data

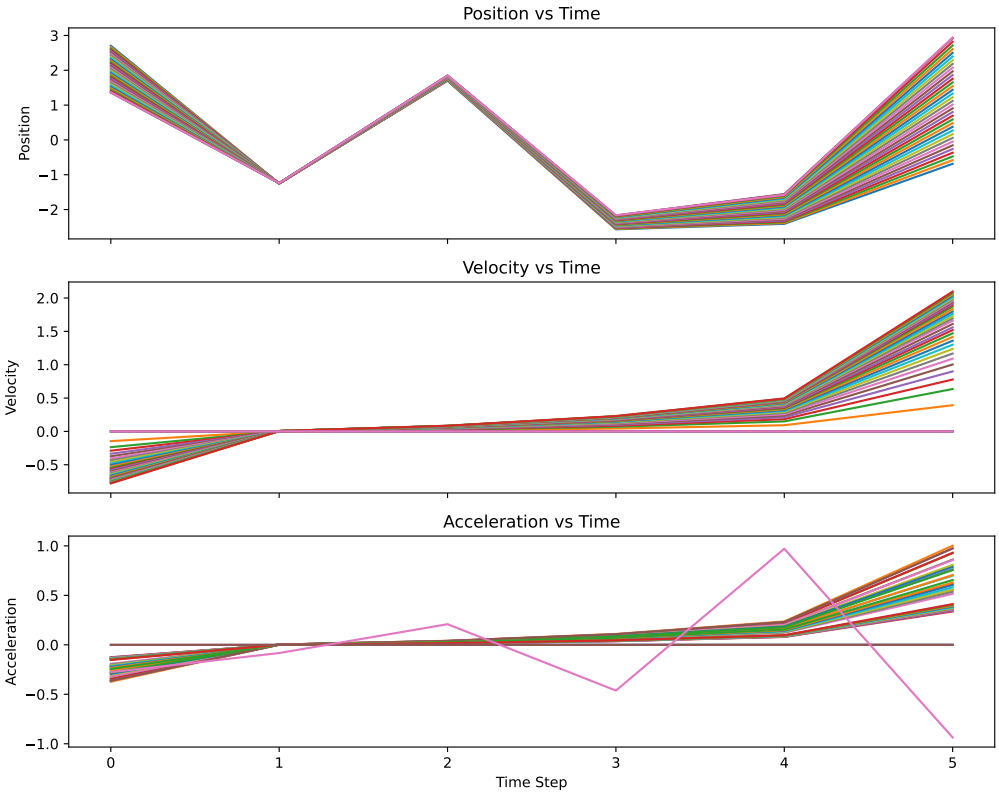


Figure A.9: Return Free Motion Trajectory Data

Bibliography

Alexi, E., Kenny, J., Atanasova, L., Casas, G., Dörfler, K., & Mitterberger, D. (2024). Cooperative augmented assembly (caa): Augmented reality for on-site cooperative robotic fabrication. *Construction Robotics*, *8*. https://doi.org/10.1007/s41693-024-00138-6

Alfieri, E., E. Seghezzi, M. S., & G. M. Di Giuda, G. M. (2020). A bim based approach for dfma in building construction: Framework and first results on an italian case study. *Architectural Engineering and Design Management*, *16*(4), 247–269. https://doi.org/10.1080/17452007.2020.1726725

Alwisy, A., Samer Bu Hamdan, B. B., & Ahmed Bouferguene, M. A.-H. (2019). A bim-based automation of design and drafting for manufacturing of wood panels for modular residential buildings. *International Journal of Construction Management*, *19*(3), 187–205. https://doi.org/10.1080/15623599.2017.1411458

Amtsberg, F., Yang, X., Skoury, L., Wagner, H., & Menges, A. (2021). Ihrc: An ar-based interface for intuitive, interactive and coordinated task sharing between humans and robots in building construction. https://doi.org/10.22260/ISARC2021/0006

Breyer, D. E., Cobeen, K., Pollock, K. J., & Schramm, D. G. (2019). *Design of wood structures—asd/lrfd* (8th). McGraw-Hill Education. https://www.accessengineeringlibrary.com/content/book/9781260128673

Cai, W., & Zou, Z. (2022). A reinforcement learning based approach for conducting multiple tasks using robots in virtual construction environments. *Automation in Construction toward Resilience: Robotics, Smart Materials and Intelligent Systems*. https://doi.org/10.22260/ICRA2022/0014

Campbell, J. (2019). *Building with timber: Paths into the future*. Laurence King Publishing.

Cascade, O. (2023). *Open cascade technology documentation* [Accessed: 2024-02-06]. https://dev.opencascade.org/

Chen, Z., Gu, H., Bergman, R., & Liang, S. (2020). Comparative life-cycle assessment of a high-rise mass timber building with an equivalent reinforced concrete alternative using the athena impact estimator for buildings. *Sustainability*, *12*, 4708. https://doi.org/10.3390/su12114708

Chitta, S., Sucan, I., & Cousins, S. (2012). Moveit![ros topics]. *IEEE Robotics & Automation Magazine - IEEE ROBOT AUTOMAT*, *19*, 18–19. https://doi.org/10.1109/MRA.2011.2181749

Cousins, S. (2012). Is ros good for robotics? *IEEE Robotics & Automation Magazine - IEEE ROBOT AUTOMAT*, *19*, 13–14. https://doi.org/10.1109/MRA.2012.2193935

Craig, J. (2021). *Introduction to Robotics Mechanics and Control, Global Edition*. Pearson Deutschland. https://elibrary.pearson.de/book/99.150005/9781292164953

D'Amico, B., Pomponi, F., & Hart, J. (2021). Global potential for material substitution in building construction: The case of cross laminated timber. *Journal of Cleaner*

Production, *279*, 123487. https://doi.org/https://doi.org/10.1016/j.jclepro.2020.123487

DOCKER, I. (n.d.). Docker documentation [Accessed: 2024-11-03]. https://docs.docker.com

DÖRFLER, K., SANDY, T., GIFTTHALER, M., GRAMAZIO, F., KOHLER, M., & BUCHLI, J. (2016). Mobile robotic brickwork. In D. REINHARDT, R. SAUNDERS, & J. BURRY (Eds.), *Robotic fabrication in architecture, art and design 2016* (pp. 204–217). Springer International Publishing. https://doi.org/10.1007/978-3-319-26378-6_15

EVERSMANN, P., GRAMAZIO, F., & KOHLER, M. (2017). Robotic prefabrication of timber structures: Towards automated large-scale spatial assembly. *Construction Robotics*, *1*. https://doi.org/10.1007/s41693-017-0006-2

FEIO, A. O., LOURENÇO, P. B., & MACHADO, J. S. (2014). Testing and modeling of a traditional timber mortise and tenon joint. *Materials and Structures*, *47*(1), 213–225. https://doi.org/10.1617/s11527-013-0056-y

GEISSDOERFER, M., SAVAGET, P., BOCKEN, N. M., & HULTINK, E. J. (2017). The circular economy – a new sustainability paradigm? *Journal of Cleaner Production*, *143*, 757–768. https://doi.org/https://doi.org/10.1016/j.jclepro.2016.12.048

GRAMAZIO, F., & KOHLER, M. (2008). Digital materiality in architecture. *Architectural Design*, *78*(4), 14–21.

HARRIS, R. (1993). *Discovering timber-framed buildings*. Shire Publications.

HART, J., & POMPONI, F. (2020). More timber in construction: Unanswered questions and future challenges. *Sustainability*, *12*, 3473. https://doi.org/10.3390/su12083473

HASSAN, R., IBRAHIM, A., & AHMAD, Z. (2023). *Timber connections: Mortise and tenon structural design* (1st ed.). Springer Singapore. https://doi.org/10.1007/978-981-19-2697-6

HERZOG, T., NATTERER, J., SCHWEITZER, R., VOLZ, M., & WINTER, W. (2012). *Timber construction manual* (6th). Birkhäuser.

KAMALI, M., & HEWAGE, K. (2015). A framework for comparative evaluation of the life cycle sustainability of modular and conventional buildings. *Proceedings of the 2015 Modular and Offsite Construction (MOC15) Summit & 1st International Conference on the Industrialization of Construction (ICIC)*.

KEEPING, M., & SHIERS, D. (Eds.). (2017). *Sustainable building design: Principles and practice*. Wiley-Blackwell. https://www.amazon.de/-/en/Miles-Keeping-ebook/dp/B076JV6N6C

KYJANEK, O., AL BAHAR, B., VASEY, L., WANNEMACHER, B., & MENGES, A. (2019, May). Implementation of an augmented reality ar workflow for human robot collaboration in timber prefabrication. In M. AL-HUSSEIN (Ed.), *Proceedings of the 36th international symposium on automation and robotics in construction (isarc)* (pp. 1223–1230). International Association for Automation; Robotics in Construction (IAARC). https://doi.org/10.22260/ISARC2019/0164

LAVALLE, S. M. (2006). *Planning algorithms*. Cambridge University Press.

LAWSON, R. M., & OGDEN, R. G. (2010). Sustainability and process benefits of modular construction. *Proceedings of the 18th CIB World Building Congress, TG57-Special Track*, 38–51.

Liu, H., Zhang, Y., Lei, Z., Li, H. X., & Han, S. (2021). Design for manufacturing and assembly: A bim-enabled generative framework for building panelization design. *Advances in Civil Engineering*, *2021*. https://doi.org/10.1155/2021/5554551

McNeel. (n.d.). Rhinoceros [Accessed:2025-01-24]. https://www.rhino3d.com/

McNeel, R. (n.d.). Grasshopper documentation [Accessed: 2024-11-05].

Mele, T. V., & many others. (2017-2021). COMPAS: A framework for computational research in architecture and structures. [http://compas.dev]. https://doi.org/10.5281/zenodo.2594510

Mitterberger, D., Ercan Jenny, S., Vasey, L., Lloret-Fritschi, E., Aejmelaeus-Lindström, P., Gramazio, F., & Kohler, M. (2022). Interactive robotic plastering: Augmented interactive design and fabrication for onsite robotic plastering. *CHI Conference on Human Factors in Computing Systems (CHI '22)*, 1–18. https://doi.org/10.1145/3491102.3501842

Mitterberger, D., Atanasova, L., Dörfler, K., Gramazio, F., & Kohler, M. (2022). Tie a knot: Human–robot cooperative workflow for assembling wooden structures using rope joints. *Construction Robotics*, *6*. https://doi.org/10.1007/s41693-022-00083-2

Overby, A. (2010). *Cnc machining handbook: Building, programming, and implementation*. McGraw-Hill/TAB Electronics.

Padilla-Rivera, A., Amor, B., & Blanchet, P. (2018). Evaluating the link between low carbon reductions strategies and its performance in the context of climate change: A carbon footprint of a wood-frame residential building in quebec, canada. *Sustainability*, *10*(2715). https://doi.org/10.3390/su10082715

Pan, M., Linner, T., Pan, W., Cheng, H., & Bock, T. (2018). A framework of indicators for assessing construction automation and robotics in the sustainability context. *Journal of Cleaner Production*, *182*, 82–95. https://doi.org/https://doi.org/10.1016/j.jclepro.2018.02.053

PickNikInc. (n.d.). Moveit documentation [Accessed: 2024-11-03]. https://moveit.ros.org/documentation/

pythonOCC Community. (2023). *Pythonocc – 3d cad/cae/plm development framework for python* [Accessed: 2024-02-06]. http://www.pythonocc.org/

Quesada, R. (2005). *Computer numerical control: Machining and turning centers*. Prentice Hall PTR. https://books.google.de/books?id=Lh5FNQAACAAJ

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., & Ng, A. (2009). Ros: An open-source robot operating system. *3*.

Robotiq. (2023, August). Start Production faster | RobotIq. https://robotiq.com/

Robots, U. (n.d.). Universal robots collaborative robots [Accessed: 2024-11-03]. https://www.universal-robots.com

ROS.org. (n.d.-a). Ros documentation [Accessed: 2024-11-03]. https://www.ros.org

ROS.org. (n.d.-b). Rviz documentation [Accessed: 2024-11-03]. https://wiki.ros.org/rviz

Rust, R., Casas, G., Parascho, S., Jenny, D., Dörfler, K., Helmreich, M., Gandia, A., Ma, Z., Ariza, I., Pacher, M., Lytle, B., Huang, Y., Kasirer, C., Bruun, E., & Leung, P. (2018). COMPAS FAB: Robotic fabrication package for the compas

framework [Gramazio Kohler Research, ETH Zürich]. https://doi.org/10.5281/zenodo.3469478

SAIDI, K. S., BOCK, T., & GEORGOULAS, C. (2016). Robotics in construction. In B. SICILIANO & O. KHATIB (Eds.), *Springer handbook of robotics* (pp. 1493–1520). Springer International Publishing. https://doi.org/10.1007/978-3-319-32552-1_57

SCHMIDT, R. J., & DANIELS, C. E. (1999, April). *Design considerations for mortise and tenon* (tech. rep.). Department of Civil and Architectural Engineering, University of Wyoming. Laramie, WY, USA.

SICILIANO, B., SCIAVICCO, L., VILLANI, L., & ORIOLO, G. (2008, November). *Robotics - Modelling, Planning and Control*. Springer London. https://doi.org/10.1007/978-1-84628-642-1

SMITH, R. E. (2011). *Prefab architecture: A guide to modular design and construction*. John Wiley & Sons, Inc.

TANNE, Y., & INDRAYANI, N. (2023). Implementation of construction automation and robotics (car) in indonesian construction state-owned enterprises: Position in project life cycle, gap to best practice and potential uses. https://doi.org/10.21203/rs.3.rs-2501558/v1

TEDESCHI, A. (2014). *Aad algorithms-aided design: Parametric strategies using grasshopper*. Le Penseur.

UNIVERSAL ROBOTS. (2025). Ur10e – collaborative industrial robot [Accessed: 2025-02-16]. https://www.universal-robots.com/products/ur10e/

WOODBURY, R. (2010). *Elements of parametric design*. Routledge.

XIAO, Y., PAN, X., TAVASOLI, S., AZIMI, M., BAO, Y., FARSANGI, E., & YANG, T. (2023, August). Autonomous inspection and construction of civil infrastructure using robots. https://doi.org/10.1201/9781003325246-1

YOUNIS, A., & DODOO, A. (2022). Cross-laminated timber for building construction: A life-cycle-assessment overview. *J. Build. Eng.*, *52*, 104482. https://doi.org/10.1016/j.jobe.2022.104482

YUAN, Z., SUN, C., & WANG, Y. (2018). Design for manufacture and assembly-oriented parametric design of prefabricated buildings. *Automation in Construction*, *88*, 13–22. https://doi.org/10.1016/j.autcon.2017.12.021

ZADEH, P., STAUB-FRENCH, S., CALDERON, F., CHIKHI, I., POIRIER, E., CHUDASMA, D., & HUANG, S. (2018, November). *Building information modeling (bim) and design for manufacturing and assembly (dfma) for mass timber construction*.

Declaration

I hereby affirm that I have independently written the thesis submitted by me and have not used any sources or aids other than those indicated.

Location, Date, Signature