Department of Mechanical and Production Engineering

Specialization Project in Robotics

Automatization of chamfering process using robot technology on charge air cooler/EGR for Vestas Aircoil A/S

June 2023

Project supervisor: Xuping Zhang
Co-supervisor: Xingyu Yang
Authors:

202102198 - Antonio Karlović
202102200 - Petar Gavran
202103839 - Adam Mariusz Kuryla

AARHUS UNIVERSITET

Contents

1 Introduction **1**
- 1.1 Background and motivation . . . 1
- 1.2 Problem statement . . . 1
- 1.3 Objectives and scope . . . 2
- 1.4 Outline of the report . . . 2

2 Literature Review **3**
- 2.1 Overview of process automation and industrial robotics . . . 3
- 2.2 State-of-the-art in robotic chamfering . . . 3
- 2.3 Relevant research on robot programming and computer vision . . . 4

3 Process Overview and System Integration **6**
- 3.1 Introduction . . . 6
- 3.2 Process overview . . . 6
- 3.3 Hardware integration . . . 6
  - 3.3.1 UR5e . . . 6
  - 3.3.2 MiR200 . . . 8
  - 3.3.3 ER-ability setup . . . 9
- 3.4 Software integration . . . 9
  - 3.4.1 ROS1 . . . 10
  - 3.4.2 UR5e & MiR200 drivers . . . 10
  - 3.4.3 System Architecture . . . 10
- 3.5 Robot description creation (URDF) . . . 12
- 3.6 Summary . . . 14

4 End-effector Design and Implementation **15**
- 4.1 Introduction . . . 15
- 4.2 Motor . . . 15
- 4.3 Mounting and connections . . . 16
- 4.4 Camera and Arduino attachment . . . 17
- 4.5 Control of the Drill . . . 18
  - 4.5.1 Python Script . . . 19
  - 4.5.2 Arduino Sketch . . . 19
- 4.6 Summary . . . 20

5 Chamfering Study **21**
- 5.1 FEM . . . 21
  - 5.1.1 Methods . . . 21
  - 5.1.2 Bodies and boundary condition . . . 23
  - 5.1.3 Settings of simulation . . . 25
  - 5.1.4 Results . . . 26
  - 5.1.5 Simulation with SPH . . . 29
- 5.2 Theoretical model and experimental study . . . 33

5.3 Summary 37

6 Optimization of UR5e Joint Configuration 38
6.1 Optimization Approach 38
6.2 Optimization Process 38
6.3 Mathematical Formulation of the Optimization Problem 39
6.4 Optimization Results and Validation 41
6.4.1 Optimization of UR5e Alone 41
6.4.2 Inclusion of the Drill 42
6.5 Inclusion of the Torques and Forces occurred during Chamfering Process 42
6.6 Summary 44

7 Computer Vision 46
7.1 Introduction 46
7.2 Eye-in-hand camera calibration 46
7.2.1 MoveIt calibration 47
7.2.2 Results and analysis 49
7.3 End-effector alignment 51
7.3.1 Camera Calibration 51
7.3.2 Checkerboard Detection and End-Effector Alignment 54
7.3.3 Validation and Error Mitigation 57
7.4 Pipe recognition 61
7.4.1 Methods 61
7.4.2 Model training and validation 63
7.4.3 Program structure 69
7.4.4 Detection of coordinates and depths 71
7.4.5 Sorting of coordinates 72
7.4.6 Perspective transformation 74
7.4.7 Saving of data 75
7.4.8 Analysis of the computer vision system's ability to recognize pipes 75
7.5 Summary 79

8 Robot Programming and Movement 80
8.1 UR5e 80
8.1.1 MoveIt Commander 80
8.1.2 Node Initialization 80
8.1.3 Publishers and Subscribers 81
8.1.4 Control of the UR5e manipulator 81
8.2 MiR200 82
8.2.1 Node Initialization 82
8.2.2 Publishers and Subscribers 82
8.2.3 The Odometry Callback Function 82
8.2.4 The Move Distance and Rotate Methods 82
8.2.5 Distance and Angle Calculations 82
8.2.6 Testing the Controller 83
8.3 Mobile Robot Manipulator 83

8.3.1 MoveIt Package Creation 83
8.3.2 Launch File Creation and System Communication 84
8.3.3 Control of the Mobile Robot Manipulator for Automation of Chamfering Process 86
8.4 Results and validation 92
8.5 Summary 93

9 Digital Twin Development 95
9.1 Introduction 95
9.2 Methodology 95
9.3 Benefits 98
9.4 Post-processing and optimization 98
9.5 Summary 102

10 Conclusion & Discussion 103
10.1 Summary of the project and its outcomes 103
10.2 Significance of the results for process automation and industrial robotics 103
10.3 Recommendations for future work 103

A Processing of coordinates of detected pipes - Python code I
A.1 Initial part of code (Importing libraries and defining paths) I
A.2 Activation of Real Sense camera II
A.3 Prediction and detection of objects with YOLO III
A.4 Determining of Cartesian coordinates of pipes IV
A.5 Initial results printed in console and default saving of YOLO model . V
A.6 Loop break and depth filtering V
A.7 Saving of data V
A.8 Sorting and Perspective transformation function activation VI
A.9 Parameters for YOLO model and end of program VI

B Calibration of camera - Python code VIII

C Sorting and perspective funtions - Python code X
C.1 Sorting X
C.2 Perspective X

D Gazebo Digital Twin codes XI
D.1 Launch file XI
D.2 UR5e Joint Replicator code in Python XI
D.3 `cmd_vel_forwarder` code in Python XII

E Digital Twin - Processing of the signal XIII

F Drill motor control - Code XV

G Optimization of UR5e joint configuration - MATLAB code XVII

H Alignment of end-effector with checkerboard - Python code XIX

I Chamfering of pipes - Python code **XXIV**

J MiR200 control - Python code **XXIX**

K Automation process of chamfering - Python code **XXXI**

Abstract

The automation of industrial processes plays a vital role in enhancing productivity and efficiency. This research focuses on developing a robotic system integrated with computer vision capabilities for automating the chamfering process of pipes. The aim is to achieve precise and consistent chamfering, reducing manual labour and enhancing the overall production workflow. The project utilizes the UR5e robot and MiR200 mobile base, programmed within the ROS1 framework on Ubuntu. The methodology includes the creation of robot descriptions, hardware and software integration, computer vision-based pipe recognition using YOLOv5, robot, and mobile base programming, coordination between physical parts and software, end-effector design with a motorized drill, and the development of a digital twin. The results demonstrate the successful implementation and integration of the robotic system, accurate pipe recognition, and reliable chamfering performance. The findings contribute to the field of process automation and industrial robotics by showcasing the feasibility and benefits of employing robotic systems in pipe chamfering applications. Future work could explore additional optimization techniques and expand the system's capabilities to other industrial processes.

1 | Introduction

1.1 Background and motivation

Vestas Aircoil A/S is a renowned manufacturer of charge air coolers, intercoolers, and cooling towers. Founded in the small Danish town of Lem in Jutland, the company has a history that dates back to 1956 when it collaborated with Vestas Wind Systems to build the first marine diesel engine charge air cooler for Burmeister & Wain.

To remain competitive in a rapidly evolving market, Vestas Aircoil recognizes the need to embrace the latest technologies and innovations. By continuously adopting advanced solutions and exploring automation, the company aims to minimize reliance on manual labor and optimize efficiency. This commitment to embracing automation is crucial in meeting the growing demands of the industry and maintaining its leadership status.

The company's drive to adapt to emerging technologies and prioritize automation is strategic and aligned with their commitment to excellence. By leveraging automation, Vestas Aircoil aims to enhance production processes, improve product quality, and reinforce its position as an industry leader dedicated to innovation and continuous improvement.

1.2 Problem statement

The main question that drives this research is whether the chamfering procedure, traditionally performed manually, can be effectively automated. This investigation aims to address the feasibility of automating the chamfering process, determine the optimal procedure for automation, and explore the benefits associated with this technological advancement.

Sub-questions that guide this study are as follows:

- Feasibility: Is it feasible to automate the chamfering procedure that has been performed manually until now? This question explores the technical aspects, considering factors such as robot capabilities, programming requirements, and system integration.
- Optimal Procedure: What is the optimal procedure for automating the chamfering process? This question focuses on determining the most efficient and effective way to carry out chamfering tasks using robotic automation, considering factors such as tool selection, motion planning, and control strategies.
- Benefits of Automation: What benefits does the automatization of the chamfering process bring? This question explores the advantages of implementing automation in chamfering, including increased productivity, improved precision, reduced labor costs, and enhanced safety for workers.

By addressing these questions, this research aims to provide valuable insights into the potential for automating the chamfering procedure, offering guidance for industrial processes seeking to optimize their operations and leverage the benefits of robotic automation.

1.3 Objectives and scope

The main objective of this research is to deliver a comprehensive report that provides a detailed understanding of the feasibility and technical aspects of automating the chamfering procedure. The report aims to present findings and recommendations that the company can utilize to scale and implement the automated chamfering process according to their specific requirements.

The scope of this study is primarily focused on the technical aspects of automating the chamfering procedure. The research will dive into various aspects such as robot capabilities, programming requirements, system integration, tool selection, motion planning, and control strategies. By analyzing these factors, the study aims to provide technical insights and recommendations that can guide the company in implementing an optimized and efficient automated chamfering process.

It is important to note that this research will not include an economic analysis of the automation process. The study recognizes that automation is a long-term investment and understands that the benefits of automation go beyond immediate financial gains. Therefore, the focus will be on the technical feasibility and optimization of the chamfering process, rather than a detailed economic analysis.

By concentrating on the technical aspects and disregarding the economic factors, this research aims to provide valuable insights and recommendations that will enable the company to make informed decisions regarding the implementation of an automated chamfering process.

1.4 Outline of the report

At the beginning of the report, the reader will be introduced to the goal of the automation procedure in the form of a process flowchart. The robot technology used in the project will be described in the following chapter, followed by a discussion on the most important component of the robot, the end-effector. The chamfering study will be examined in detail, incorporating both theoretical and FEM approaches for analysis. The subsequent chapter will go into the programming of robot movements in space. The critical section on computer vision will be divided into camera calibration, end-effector alignment, and pipe recognition. Lastly, the development of the digital twin will be explored. Each section will include a thorough analysis of the results. The report will conclude with a comprehensive summary.

2 | Literature Review

2.1 Overview of process automation and industrial robotics

Automation can significantly enhance a company's operational efficiency, reduce costs, and mitigate human error, thereby fostering a competitive advantage and driving business growth. However, automation also stirs fear of job displacement among workers, particularly low-skilled ones, affecting job satisfaction and potentially impacting productivity [1]. Although automation can initially displace employment, indirect gains in other industries and increased aggregate demand can counteract this effect [2].

Industrial robots have been the subject of much discussion due to their potential impacts on labor markets and job displacement. Acemoglu and Restrepo [3] conducted a comprehensive analysis on this issue, examining data from US labor markets within commuting zones from 1990 to 2007. They found that an increase in industrial robots corresponded to a reduction in the employment-to-population ratio by approximately 0.2 percentage points. Furthermore, it led to a decrease in wages by between 0.25 and 0.5 percent, with the most significant effects being noted among routine manual occupations and those with low education.

On the other hand, Brynjolfsson and Mitchell [4] provide a different perspective, focusing on the abilities of machine learning and industrial robotics. They argue that not all tasks are subject to automation, specifically highlighting complex perception and manipulation tasks, creative intelligence tasks, and social intelligence tasks as ones that machines are yet to master. This suggests that certain jobs may be relatively safe from the reach of automation, at least in the immediate future.

In a similar vein, Chui et al. [5] explore the potential for automation across various occupational tasks. They assert that more occupations are likely to change than be completely automated in the near future, a viewpoint that presents a less dire outlook than outright job displacement. They further emphasize the importance of considering the broader benefits of automation that extend beyond labor substitution.

In summary, automation and industrial robotics offer companies enhanced efficiency and cost reduction. However, there are concerns about job displacement, especially among routine manual occupations. Not all tasks can be automated, leaving room for job evolution rather than a complete replacement. Therefore, it is crucial to consider the broader benefits and changes that automation can bring.

2.2 State-of-the-art in robotic chamfering

The connection between drilling operations and the usage of robot arms use to appear in multiple research papers. The machining procedure including material removal is

related to multiple problems, which can affect the usage of robot arms in the process. Most of the researched literature describes problems with the stiffness of the robot during the procedure [6],[7],[8],[9]. Some of the researchers like J.R.V. Sai Kiran [7] provided a comparison between the usage of industrial robots and classic CNC machines, however, due to their conclusions, most studies are made on older robots, which are far more advanced nowadays, which could provide better stiffness.

Moreover, some studies developed a theoretical model for defining forces and torques appearing during the process of drilling. One of those studies was conducted by Sebastien Garniera, Kevin Subrina, and Kriangkrai Waiyaganb [6], which aimed to be part of an accuracy study on drilling with the robot.

However, most studies are not touching the topic of chamfering process of ends of pipes and they are not developing techniques to conduct them. Therefore, other promising techniques like Finite Element Methods using different approaches might be used. There are no existing research papers that could provide answers to resultant forces and as well torques and impact on end-effector, except stiffness and accuracy problems. However, similar techniques using for example SPH elements were already explained as their challenges. One of the authors V. Gyliene in his paper [10] described methods to control simulation with the usage of mesh-less techniques, nonetheless author did not explain the usability of the method and the outputs that could be used from it. Moreover, the paper focuses mainly on drilling matter rather than chamfering of pipe, which can have a completely other nature.

2.3 Relevant research on robot programming and computer vision

The merging of robot programming and computer vision is the subject of extensive research, especially in the context of process automation. This section provides an overview of some of the interesting studies in this area, with a focus on those relevant to the automation of pipe chamfering processes on coolers.

The work of [11] developed a robust and precise monocular vision system for robotic drilling. The system is designed to increase the accuracy of drilled hole positions through error compensation, a concept that could be applied to the chamfering process in coolers. The authors also presented an elliptic saliency-snake contour extraction algorithm for robust and highly accurate hole reference detection, which could be useful for identifying and locating pipes on heat sinks.

Building on the work of [11], [12] proposed a new approach to robotic drilling using a monocular vision system. The authors developed a new algorithm to improve the accuracy and efficiency of the drilling process. This research is particularly relevant because it highlights the potential of advanced algorithms in improving the performance of vision systems, which could be applied to the automation of chamfering processes on refrigerators.

In another study, [13] proposed an approach for robot programming using a combination of computer vision and deep learning techniques. The researchers developed

a system that can recognize and locate objects in a 3D environment, enabling the robot to perform tasks autonomously. This research is relevant as it demonstrates the potential of combining computer vision and deep learning in robot programming.

Paper [14] focused on improving the efficiency of robotic drilling also using a monocular vision system. The authors proposed a new method for system calibration that significantly reduces the time and effort required for the process. This research is relevant because it shows the potential of system calibration in improving the efficiency of automated processes, which could be useful in the case of this automation as well.

In conclusion, the literature suggests that the use of monocular vision systems in robotic drilling has significant potential for automating complex tasks such as the chamfering process on coolers. However, further research is needed to fully realize this potential and to address the challenges associated with implementing these technologies in an industrial setting.

3 | Process Overview and System Integration

3.1 Introduction

This chapter provides an in-depth exploration of the intricate process of robot description and integration, which serves as the fundamental framework for automation project. The chapter encompasses three key aspects: the creation of a virtual representation of the robot using Unified Robot Description Format (URDF) files, the physical connection and integration of the hardware components, including the UR5e robot and MiR200 mobile base, and the pivotal software integration that acts as the vital link between the real and virtual world, as well as between the various components of the system.

3.2 Process overview

This chapter provides an overview of the process involved in the automation project and presents a comprehensive flowchart (Figure 3.1) that illustrates the key steps and components. The flowchart serves as a visual representation of the project's methodology, guiding the reader through the sequential stages of the automation process. By presenting this flowchart at the beginning of the report, readers will gain a clear understanding scope of the project and the organization of the subsequent chapters. This introduction sets the stage for a detailed exploration of each step, enabling readers to follow along and comprehend the project's progression.

3.3 Hardware integration

The hardware integration phase of the project involved connecting the physical components, including the UR5e robot and MiR200 mobile base, to form a cohesive automation system. This section provides an overview of the hardware integration process, taking into account the unique considerations and challenges encountered.

3.3.1 UR5e

To comprehend the hardware integration process, it is crucial to familiarize with the physical components of the UR5e robot (Figure 3.2). The UR5e is a robotic arm designed for precise and controlled movements. Very often it is used in educational institutes since it is easy to program and with excellent flexibility, it is perfect for light to medium-weight tasks. It consists of several key parts, including:

- **Base**: The base provides stability and acts as the foundation for the robot arm's movement. It is securely mounted within the workspace to ensure reliable and accurate operation.

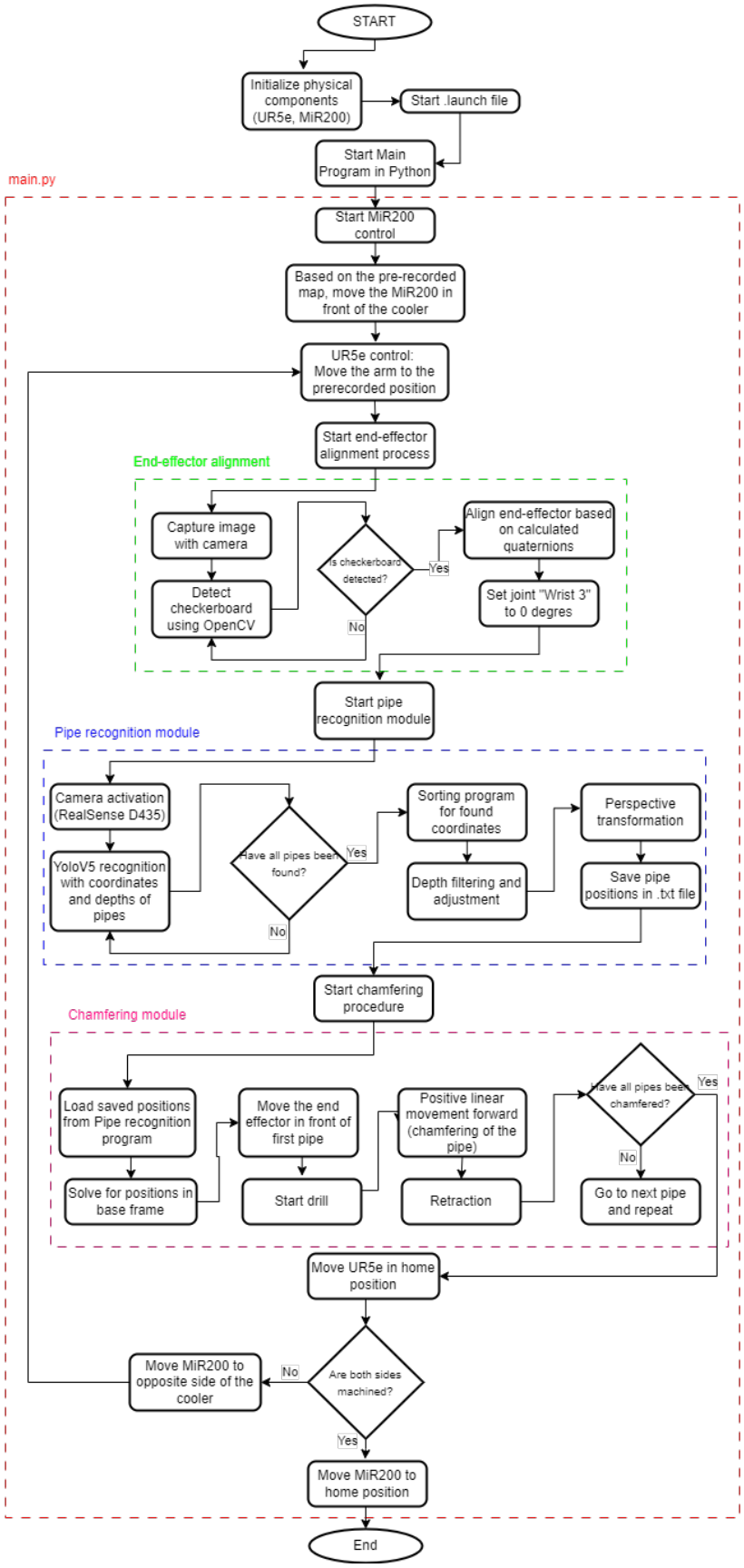


Figure 3.1: Process flow chart

- **Robot Arm**: The robot arm comprises several interconnected links and joints that enable multi-axis movements. Each joint is equipped with sensors and actuators to facilitate precise control and feedback.
- **Gripper/End-Effector**: The gripper or end-effector is the tool attached to the robot arm that performs the chamfering operation. It can be customized according to the specific requirements of the process. The tool was created for this project by combining several parts, which will be discussed in detail in Chapter 4.

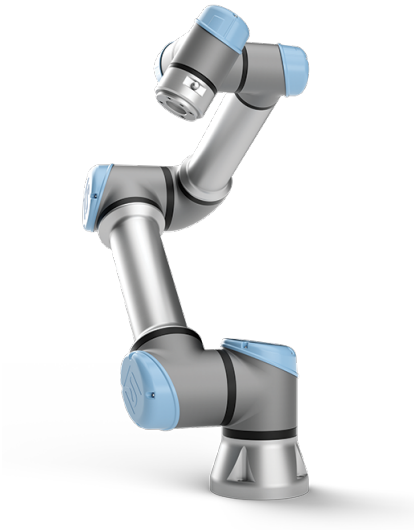


Figure 3.2: UR5e [15]

3.3.2 MiR200

Similarly, understanding the physical components of the MiR200 mobile base (Figure 3.3) is essential for the hardware integration process. The MiR200 is an autonomous mobile robot designed for flexible transportation and navigation. Its key physical parts include:

- **Base**: The mobile base provides the platform for the MiR200's movement. It is equipped with wheels or tracks and sensors for navigation and obstacle avoidance.
- **Control Unit**: The control unit houses the necessary electronics and processing capabilities to facilitate autonomous operation and communication with other system components.
- **Sensors**: The MiR200 is equipped with various sensors, such as laser scanners and cameras, which enable environmental perception and ensure safe navigation within the workspace.
- **Power Supply**: A robust power supply system ensures uninterrupted operation and sufficient power for the mobile base and its components.

Figure 3.3: MiR200 Mobile Manipulator [16]

3.3.3 ER-ability setup

In typical ER-ability setups (Figure 3.4), the MiR200 mobile base and UR5e robot are integrated using an ER-ability box, allowing for smooth communication and coordination between the two. However, due to early complications in ER-ability connections, it was decided that an independent usage approach for the MiR200 and UR5e was necessary. Although they function independently, their integration is carefully coordinated to ensure smooth operation during the process.

Figure 3.4: ER-ability [17]

3.4 Software integration

The software integration phase involved the integration of the robot control software and computer vision system using ROS1 on the Ubuntu platform. This section

elaborates on the steps taken to integrate the software components, including the implementation of `mir_driver` and `ur_drivers`.

3.4.1 ROS1

The Robot Operating System (ROS1) framework was set up on the Ubuntu system to establish a unified environment for software integration. ROS provides a flexible and modular architecture for developing and controlling robotic systems. This integration ensures efficient communication and coordination between the various software components.

Ubuntu is a highly-regarded open-source operating system widely utilized in the robotic community. Based on the Linux kernel, Ubuntu provides a stable and secure platform for robotics applications. Its user-friendly interface and extensive software ecosystem make it a preferred choice for developers and researchers in the field of robotics.

3.4.2 UR5e & MiR200 drivers

To facilitate the integration of the MiR200 and UR5e with the ROS framework, the `mir_driver` and `ur_drivers` were implemented. These software modules act as interfaces between the hardware components and the ROS system, enabling communication and control. The `mir_driver` enables communication with the MiR200 mobile base, while the `ur_drivers` establishes communication with the UR5e robot. These drivers allow for the exchange of commands, sensor data, and status information between the robot control software and the hardware components.

3.4.3 System Architecture

To provide a comprehensive understanding of the software integration process, a system architecture diagram is included, see Figure 3.5. This diagram illustrates the connections between the physical components (UR5e, MiR200), the router used for communication, the PC running the ROS system, and end-effector components. It visually represents the flow of information and the interaction between the different components, enhancing clarity and comprehension.

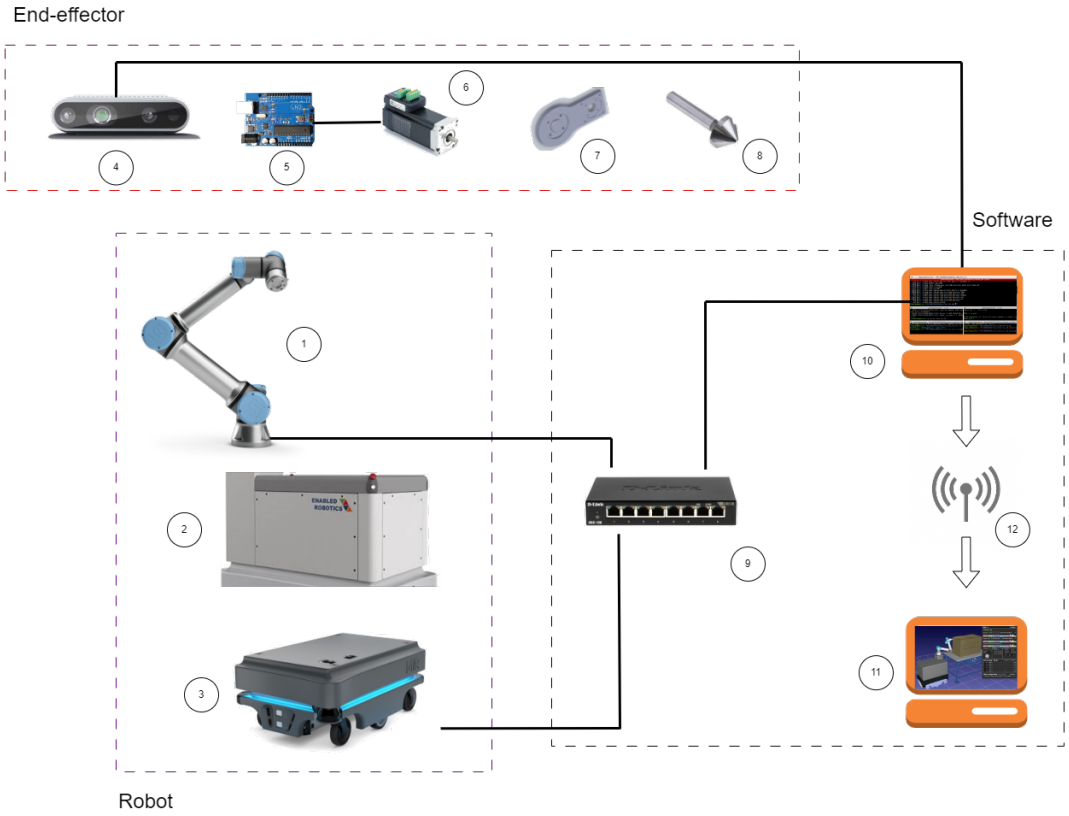


Figure 3.5: Diagram portraying connection between physical and software parts

Following parts can be seen in Figure 3.5:

- Robot:
  1. UR5e
  2. ER-ability
  3. MiR200
- End effector:
  4. Intel RealSense D435
  5. Arduino UNO board
  6. iHSV60-30-40-48 AC Servo Motor
  7. Motor holder
  8. Chamfering tool
- Software:
  9. D-Link Unmanaged Network-switch
  10. Main Ubuntu machine with launch file
  11. Second Ubuntu machine for Digital Twin
  12. ROS Master connection point

3.5 Robot description creation (URDF)

The Universal Robot Description Format (URDF) is an XML format that is widely used in the Robot Operating System (ROS) to represent robot models. It is a standard format that allows a suitable description of the robot's physical properties, including its kinematic and dynamic properties, visual representation, and collision geometries [18].

URDF originated from the ROS community and has since become an indispensable standard in the field of robotics for describing robots. It is a human-readable format that enables easy editing and understanding of the robot model. The format is hierarchical, with each link and connection defined relative to its parent [19].

A URDF file is built from a series of XML tags describing the robot's different assembled parts. The two main elements of a URDF file are links and joints. The links describe the physical properties of the robot components, such as their shape, size, and mass. Joints define the relationship between links, including their type (e.g. fixed, revolute, prismatic) and their limits (e.g. range of motion, speed, effort) [20].

The integrated mobile manipulator in this study was modeled using URDF. The robot consists of several components, each individually modeled and added to the URDF file.

The MiR200 base and its wheels, the ER box, and the UR5e with its six joints and links were all modeled in the URDF file. Each component was modeled with its specific physical properties, including mass and dimensions.

Special attention was given to the drill and camera holder attached to the end effector of the UR5e manipulator. Despite their irregular shape, they are carefully modeled in the URDF file, using previously prepared CAD models. This ensured that their physical presence was accounted for in the robot model, enabling accurate collision detection and avoidance during trajectory planning and execution.

The components are then interconnected with the appropriate types of joints. For example, UR5e and MiR200 are connected through a fixed joint in the ER box, which means that their relative position and orientation are fixed, while all joints of the UR5e robot are connected using revolute joints, which means that they can relatively change position and orientation.

The URDF file was subsequently validated. Tools such as `check_urdf` were used to check for errors in the URDF file. Additionally, the `urdf_to_graphiz` tool was used to generate a visualization of the tree of links and joints in the URDF file, which was then converted to a PDF using Graphviz for easier portrayal and it is represented in Figure 3.6.

Finally, the robot model was visualized in RViz, which can be seen on Figure 3.7, to confirm that the components were correctly linked and the dimensions were accurate.

URDF is not only a tool that would facilitate the visualization of the robot model but also played a key role in simulations and actual path planning, so it is probably

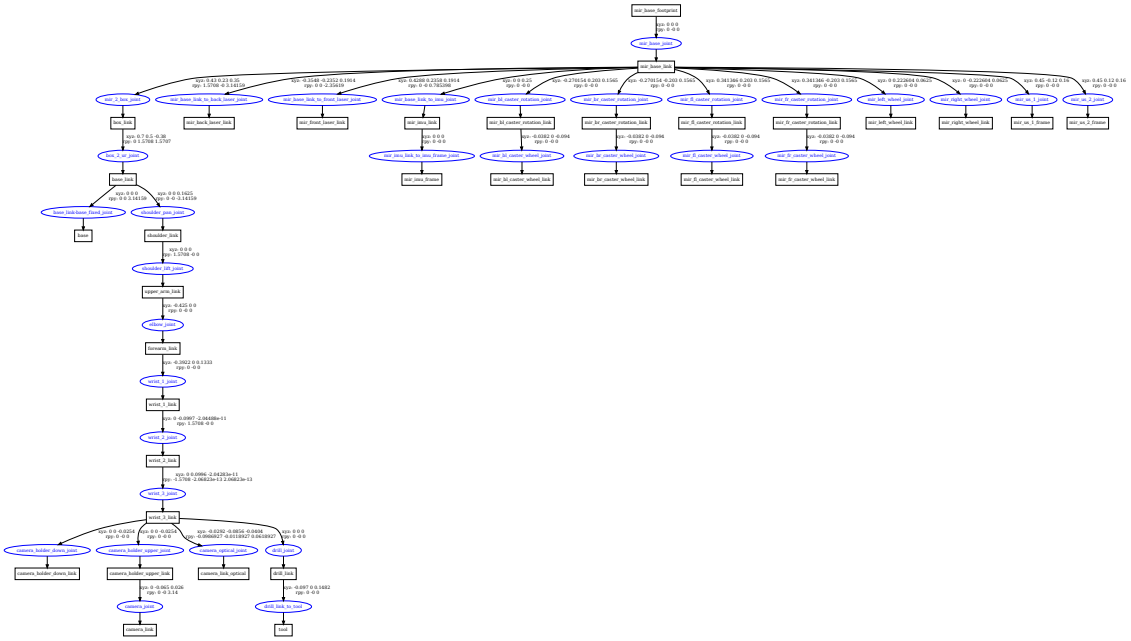


Figure 3.6: Visualization of the URDF tree

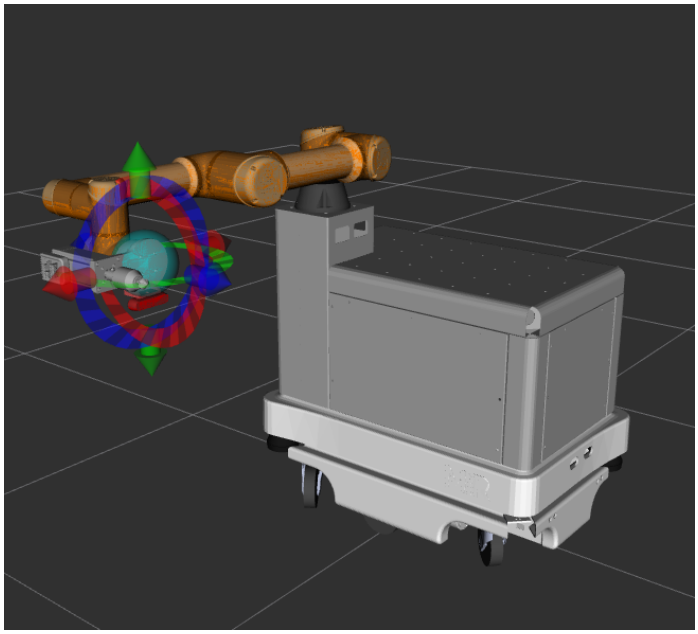


Figure 3.7: Visualization of the robot model in RViz

the most important part because it is the starting point from which the entire process is built. The 3D model of the robot derived from the URDF file was also used in Gazebo for the digital twin simulation. In MoveIt!, a motion planning framework, URDF is used for collision detection, trajectory planning, and execution [21].

This process of creating and integrating the URDF file provided a faithful digital representation of the mobile robot manipulator, enabling efficient planning and reliable

execution of tasks.

3.6 Summary

The process of description and integration of the robot forms the starting part of the automation project. Three key aspects are covered: creating a virtual representation of the robot using Unified Robot Description Format (URDF) files, physical connection and integration of hardware components, and software integration that acts as a link between the real and virtual world, as well as between different system components.

The hardware integration phase involved connecting the physical components, including the UR5e robot and the MiR200 mobile base, to form a cohesive automation system. These components are integrated using an ER-capability box, allowing communication and coordination between the two.

The software integration phase involved the stable integration of the robot control software and the computer vision system using ROS1 on the Ubuntu platform. `Mir_driver` and `ur_drivers` are implemented to facilitate the integration of MiR200 and UR5e with the ROS framework, enabling communication and control.

Furthermore, URDF was used to create a virtual representation of the robot. Each component of the robot is modeled in a URDF file. The URDF file is validated and the robot model is visualized in RViz to confirm that the components are correctly connected and that the dimensions are correct so that it can be properly used in the further process.

4 | End-effector Design and Implementation

4.1 Introduction

In order to conduct chamfering process appropriate end-effector has to be designed with specific requirements.

The first requirement is regarding chamfering tool, which can be any electric motor that is capable of conducting chamfering on PLA tubes having an attached hexagonal chamfering tool. Additionally, the motor with the tool has to be held by the robot arm by adequate attachment. Moreover, the RealSense camera has to be attached to the end-effector to proceed with camera recognition. Lastly, the Arduino module, which is controlling the Servo motor can be either placed close to the motor at the arm or placed in the control box above MiR200.

4.2 Motor

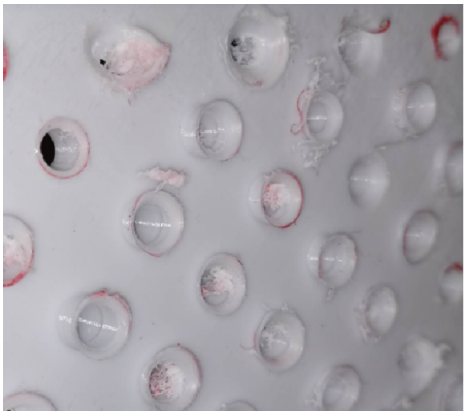
Since, UR5e has a low allowed payload of 5 [kg], as it is produced for educational rather than industrial purposes [22], not only cooler and pipes that were used for experimental chamfering had to be made from a softer material, but also used motor had to be light with enough torque and power for prototyping purposes. Based on previous studies [22] NEMA 24 servo motor was chosen with specifications shown in Table 4.1.

Model	NEMA 24
Power	400W
Type	Integrated Servo Motor
Torque	3.5Nm
Weight	1.8kg
Length	140mm
Rated speed	3000RPM

Table 4.1: Specifications of NEMA 24 Integrated Servo Motor

The motor was tested on both Copper-Nickel pipes and PLA with 3000 [RPM] (speed was described as optimal in previous research [8]) and it managed to chamfer Copper Nickel pipes with a feed rate of 0.05 $[\frac{mm}{rev}]$ [22] and with PLA pipes it managed to chamfer 5 [mm] sticking pipe within 1 [sec]. The resulting chamfered pipes can be seen in Figure 4.1.

a) Copper Nickel 90-10



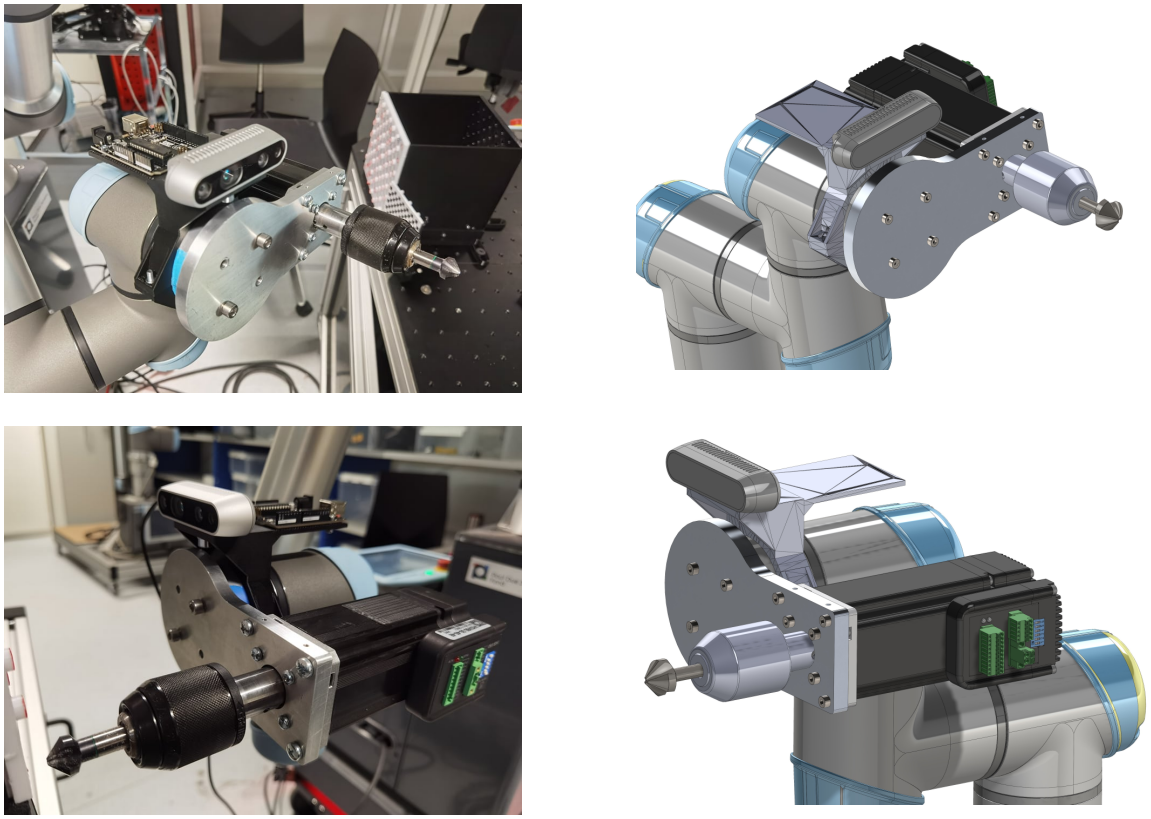
b) PLA

Figure 4.1: Chamfered pipes

4.3 Mounting and connections

The motor was mounted parallel to the last link of the robot arm as shown in Figure 4.2 with the specially designed plate. The main reason why the motor was mounted offset from the main axis of the link and not at the same axis at the center is due to the placement of heat sinks of the motor, which appeared at the back, and also due to the simplification of the design. Placement of the drilling motor in this manner could cause additional twisting moments produced by the weight of the motor and its components and impact the last joint of the robot arm. However, due to lower weight and testing of robot movements in multiple positions with mounted equipment, no issue was faced.

Since the motor was equipped with a shaft and key slot on it and the drill chuck had an internal thread, a connection adapter was produced, which can be seen in both Figure 4.2 to fix the issue.



a) Physical b) CAD

Figure 4.2: Model of end-effector

4.4 Camera and Arduino attachment

The last part of the gripper design is including attachment of the camera and Arduino to the last link of the robot arm.

Initial design

The initial design of the placement of Arduino and Camera was different from the final one. First of all, Arduino was supposed to be placed next to the PLC of UR5e, however, due to frequent testing of the motor and reduction of the length of cables between Arduino and motor board was placed closer to the motor at the last link. Moreover, the camera was supposed to be attached to the holder of the motor with an additional plate and mounted anti-vibration bushing (Figure 4.3). The design was changed based on the reduction of vibration by the placement of the camera further from the motor and attachment plate and also to align it with the last link of the robot arm.

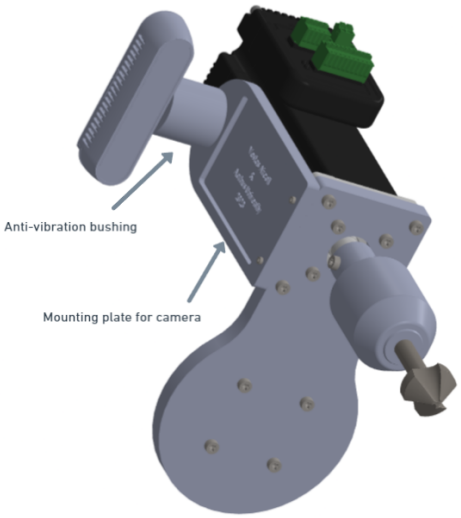


Figure 4.3: Initial model of end-effector

New design

New improved design incorporated previously described changes. The new holder for the camera and Arduino can be seen in Figure 4.4.

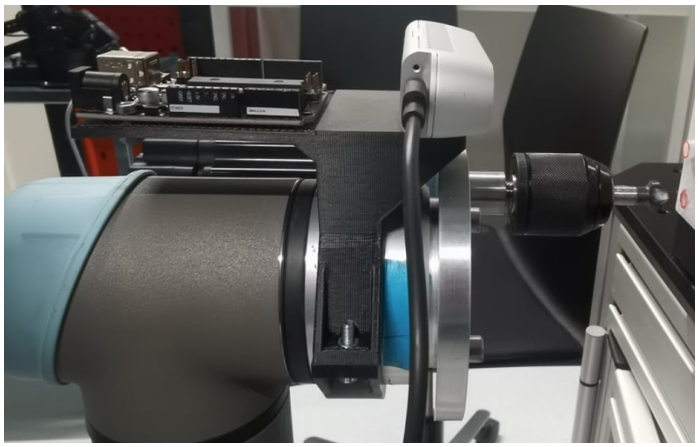


Figure 4.4: New camera holder

4.5 Control of the Drill

The control system for the iHSV60-30-40-48 AC Servo Motor involves two main parts: a Python script that serves as an interface for the user and a sketch uploaded to the Arduino board, which directly controls the motor.

4.5.1 Python Script

The Python script in Appendix F, establishes a serial connection with the Arduino board to send commands for controlling the motor's speed. It starts by uploading the Arduino sketch to the board using the Arduino Command Line Interface (CLI). It performs the following tasks:

1. Compiles and uploads the Arduino sketch using the Arduino CLI. The path to the .ino file, the port where the Arduino is connected, and the fully qualified board name are passed as parameters. The function `run_arduino_cli()` is responsible for these operations. It uses the subprocess module to execute the compile and upload commands in the system shell. If any errors occur during these operations, they are printed to the console.
2. Once the sketch is successfully uploaded, the script establishes a serial connection with the Arduino board.
3. It then defines a function, `set_motor_rpm()`, which sends a speed command to the Arduino. The speed command is a string that represents the desired RPM (revolutions per minute), followed by a newline character.
4. Finally, the script sends an example command to start the motor at 3000 RPM, waits for the user to press enter, and then sends another command to stop the motor.

4.5.2 Arduino Sketch

The Arduino sketch Appendix F, reads the commands sent over the serial connection and controls the motor accordingly. The sketch uses the `PUL` and `DIR` pins to control the motor's speed and direction, respectively. The operations performed by the sketch are as follows:

1. In the `setup()` function, the `PUL` and `DIR` pins are set as OUTPUT. The serial communication started with a baud rate of 9600.
2. In the `loop()` function, the sketch constantly checks if a complete command string has been received. If so, the sketch converts the command string into an integer to get the desired RPM. It then calculates the pulse delay required to achieve this speed.
3. If the RPM is not zero, the sketch sets the rotation direction (the `DIR` pin) based on whether the RPM is positive or negative. It then uses the `PUL` pin to control the speed of the motor, by turning it on and off at the calculated delay.
4. The `serialEvent()` function is called whenever data is available to read on the serial port. It reads the incoming characters one by one and appends them to the command string. When a newline character is encountered, it sets the `stringComplete` flag to true, indicating that a complete command has been received.

The wire schematic of the drill that can help understand the code and control can be seen in Figure 4.5.

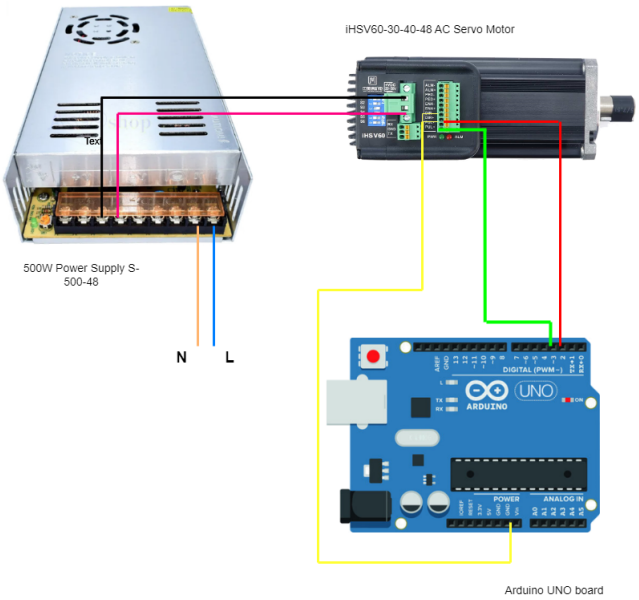


Figure 4.5: Drill wire diagram

4.6 Summary

Based on the initial design [22] and provided equipment new improved design of the end-effector was constructed and improved with time to provide the best quality and robustness of the end-effector and assure that all of the components can withstand reaction forces as well as provide full functionality for conducting process.

5 | Chamfering Study

The definition of properties including resultant forces and torques within the chamfering process is an important part of the study. Previous studies were conducted on copper nickel pipes using a rigid body as a drill and pipe defined as a solid body [22]. A finite element study however was conducted based on different material than was used in the testing setup. In the case of testing abilities to chamfer with UR5e, the PLA (Polylactic acid) model of the cooler with pipes was used, therefore properties of the material were significantly changed impacting resultant forces as well moments.

5.1 FEM

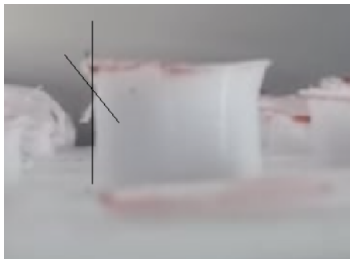
5.1.1 Methods

There are multiple approaches to conduct an explicit dynamic study on the chamfering process of pipes. Due to extremely localized deformations and numerical difficulties coming from the nature of drilling or chamfering appropriate methods might be used. In most cases, the Lagrangian method is used in which nodes follow material points. However, methods such as Arbitrary-Lagrangian–Eulerian (ALE) or meshless method of using Smoothed-Particle Hydrodynamics (SPH) for better predictions are used [23]. Moreover, the choice of explicit or implicit method is another criterion for setting up a simulation. Since chamfering is fast and requires very often longer computational time, it is more convenient to use explicit engine [24].

Nevertheless, the type of part and section used to define the chamfered pipe is an important aspect, since it might be modeled as a solid body, shell element, or build from meshless particles. In the case of shell, element simulation is less accurate due to decreased number of elements in the thickness that can be provided by the solid element, although it provides much faster calculation time and helps very quickly to establish resultant forces and torques for multiple values of feed rates and rotational speed of the spindle.

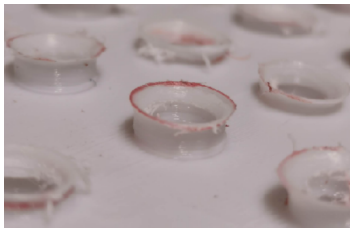
Nature of chamfering (Choice of methods)

Choice of methods, including types of the body for the pipe was mainly based on experiments on PLA cooler conducted by chamfering of multiple pipes with provided motor with 3000 [RPM] rotational speed and feed rate of 0.15 $[\frac{mm}{rev}]$.



a) PLA-side

b) Copper-Nickel



b) PLA-Top

Figure 5.1: Chamfered pipes

At each of the Figures in 5.1 it might be noticed that the nature of chamfering pipes tends to be different than in bore drilling situations. In chamfering, especially of PLA, seems that material gets removed at the point when failure appears from tensile stress after too big a plastic deformation of pipe rather than erosion. Moreover, it can be noticed in Figure 5.1 b) that cracks appear between multiple sections of the deformed section pipe rather than at the bottom of it. This is coming from the nature of the material and the slower chamfering process due to the limitations of the used motor. Since the material is ductile (properties of PLA taken from [25]), especially polylactic acid, therefore, it can withstand a higher degree of plastic deformation.

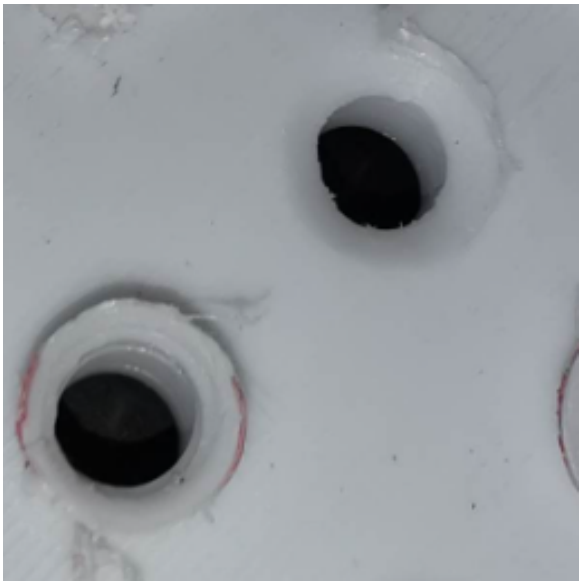


Figure 5.2: Partially chamfered pipe (lowered) and finished pipe

In Figure 5.2 it can be spotted that material removal is appearing when the hexagonal tool reaches the holding plate. At this point constantly elongating material cannot withstand further plastic deformation due to the stiffer connection between plate and pipe therefore just before fracture, non-uniform plastic behavior is displayed with the necking of the pipe section. After reaching the fracture point one big deformed chip of material is removed by the drill head leading to more uniform chip removal while progressing into the plate.

This brings the conclusion that the failure of the material does not appear until

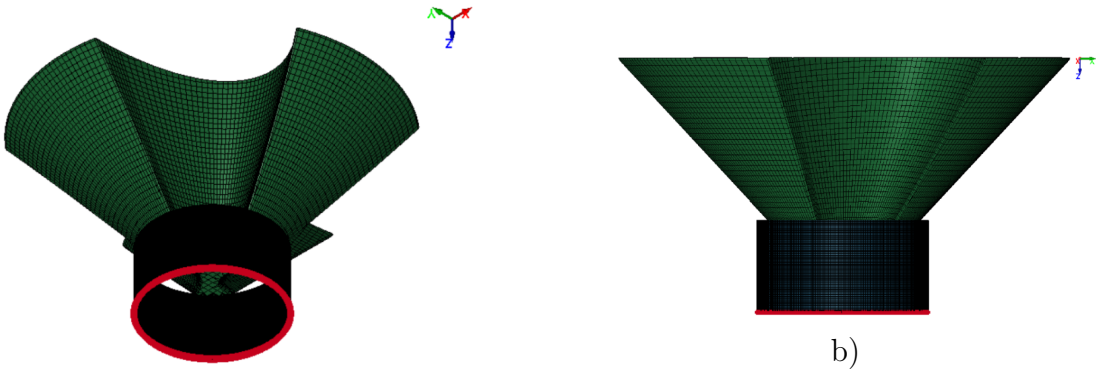
reaching the plate with the tool. It has to be said that this phenomenon appears only in perfect condition when the center of the chamfering tool coincides with the center of the chamfered pipe when chamfering tool is offset by a small distance from the center, therefore failure of the material is appearing much faster and in this case, chips of drilling are more likely to appear and is less predictable. Though, when perfect alignment of the tool is achieved it might be considered that the tool is conducting plastic deformation of the pipe by pressurizing the internal section of it and creating a cone-like shape at the contact. The described phenomenon not only can appear from the ductile nature of PLA material but also from the specification of the tool, which might be dull.

This observation helped to assume that FEM simulation could be simplified significantly by modeling a pipe as a shell instead of a solid element. In this way, the waiting time from at least 20 hours with bigger time steps, while modeling pipe as solid with elements in the thickness was reduced to a max 3 hours, while modeling pipe with fine mesh as a shell with smaller time steps. It is worth mentioning that this is not a perfect approach, since there is always some erosion in thickness appearing during the process, however, this simplified model can provide estimated values of reaction forces as well as drilling torque based on previously described phenomenon. Nevertheless, while using different materials, for example, Copper-Nickel, provided output might be different from real case scenarios, and different methods, for example, the SPH method might be used (described in the next sections).

All explicit simulations were conducted and examined with LS-Dyna software [26].

5.1.2 Bodies and boundary condition

There are two bodies (hexagonal drill and pipe) involved in the simulation, which have their specifications. The lower number of elements in the simulation might reduce calculation time, especially if some segments of them are not an important part of the mentioned simulation. The chamfering head was simplified in correlation with the previous study [22] to reduce the number of elements for the body, therefore only the section that is primarily in contact with the pipe was kept. The pipe section is also short due to the reduction of elements in length. In this way calculating time could be shortened. The drill is modeled as a rigid body and includes material properties of High-Speed Steel [27]. Moreover, the drill was modeled as a combination of shells due to provided tool in LS-Prepost [28] from the `Auto-mesh` function. LS-Prepost could not build up a model based on solid elements, therefore drill and its surface is built up from shells. Two degrees of freedom are including the movement of the drill. First, linear translation of drill towards pipe (Z translation) and second, rotation around Z axis to simulate true drilling situation. The rest of the degrees of freedom for the drill are fixed.



a) b)

Figure 5.3: Boundary conditions

The pipe was modeled as a body (Figure 5.3) prepared for plastic deformation with the usage of a simple material model (Plastic Kinematic: in LS-Dyna number 3, see Figure 5.4). The pipe has fixed support at the borderline segment at the bottom to fix all degrees of freedom and bring properties of pipe being held by a plate in real case scenario (Figure 5.3).

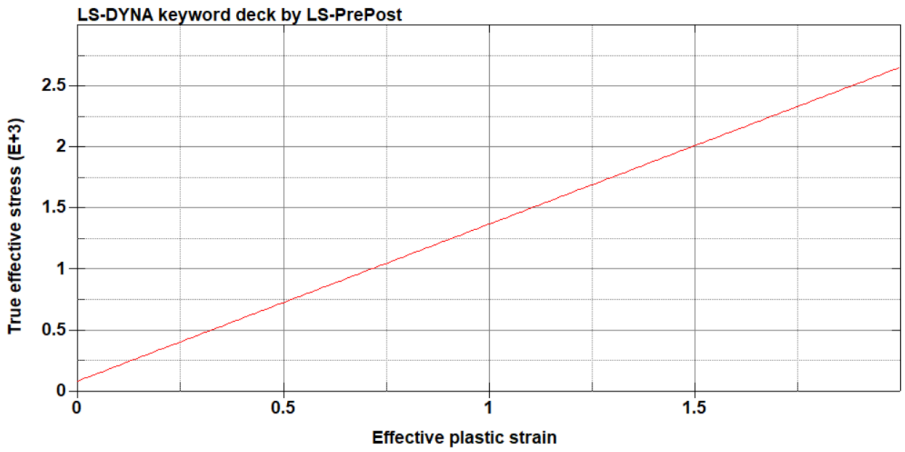


Figure 5.4: Property of material for pipe: True effective stress in MPa and Effective plastic strain

Mesh for each body was created based on their purpose and reduction of computational time. It might be noticed that the mesh of the drill (Figure 5.3) is more coarse than the mesh of the pipe. This comes from the fact that the drill is not part of the study and since it is a rigid body it is not subjected to deformation, therefore mesh size can be increased to decrease computational time. The maximum mesh size for the drill was set to 0.5 [mm]. From the other side, the mesh of the pipe must be as fine as it is possible, therefore element size should not cross 0.003 [mm]. Value was optimized since when it was decreased, the simulation tended to crash due to a high number of nodes. A pipe contains only rectangular elements within the mesh (Figure 5.5), while a drill tool is a combination of both since it has a more complex shape

and sharp edges, which for accuracy reasons should be created from triangle elements [29].

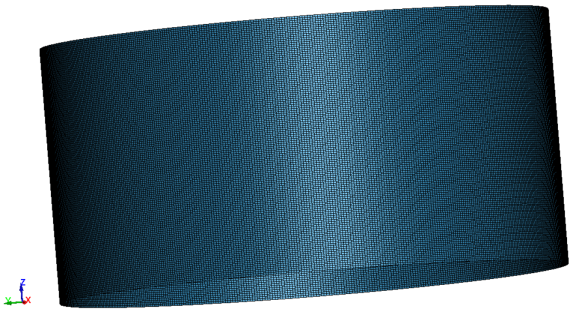


Figure 5.5: Mesh of the pipe

5.1.3 Settings of simulation

The simulation was set up in both ways, first including only mechanical properties and also including both mechanical and thermal properties due to the influence of generated heat during the chamfering process on the properties of PLA. However, due to the cost of computational time for processing thermal properties, the model was simplified to include only mechanical ones with assumptions that outputs would not be influenced significantly.

The time of simulation was set up to be 0.2 seconds with a number of steps of 10000. The number of steps was defined based on multiple tries and the stability of the simulation, which started to give good results after passing 8000 steps. Moreover, a rotational speed of N=3000 [RPM], assuming that the motor operates at standard capacity, see Figure 4.1. Within t=0.2 second drill head is translated by L=1.5 [mm] towards the pipe, giving after calculations feed rate of f=0.15 $[\frac{mm}{rev}]$. A value of 1.5 [mm] of the traveling distance of the drill was established based on the testing procedure of the motor on chamfering pipes and previous studies [22]. Feed rate per revolution can be found from the relation between traveled distance, the rotational speed of the drill, and time-based on Equation 5.1.

$$f = \frac{L \cdot 60}{t \cdot N} \tag{5.1}$$

Moreover, to decrease the time of simulation mass scaling was used in the simulation of time step -1e-6 [30].

Contact between bodies was defined as `Automatic Surface to Surface`, where the surface of the drill includes the whole body and the surface of the pipe includes a segment of elements of the shell, which is the upper part of the chamfered pipe. Moreover, static and dynamic friction coefficients were estimated and determined to be equal respectively 0.4 and 0.3 based on literature study [31].

5.1.4 Results

After establishing the simulation, resultant forces and drilling torque were determined based on contact output.

First of all, the simulation was compared visually to the experimental chamfering of pipes.

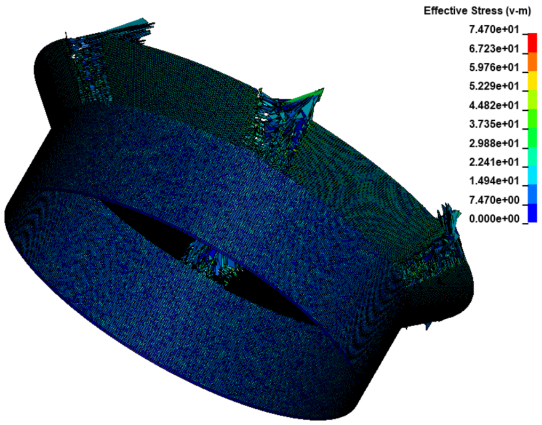


Figure 5.6: Shape of pipe at the end of simulation

It might be noticed that pipe after the procedure at Figure 5.6 follows a real case scenario and achieves the same results, which were established at the beginning of the examination, see Figure 5.1. It might be noticed that there are multiple spots between segments of deformed mesh, which failed due to plastic strain. The same situation could be noticed, while looking at the chamfered Copper-Nickel pipe, see b) in Figure 5.1, where between deformed segments of pipe cracks were appearing. Moreover, one more phenomenon is appearing during the result investigation. As can be seen at point a in Figure 5.1, between a deformed section of pipe and lowered part there is no smooth deflection, meaning that between the axis of deflected material and the initial pipe, there is no smooth corner, but very sharp one, see Figure 5.7.

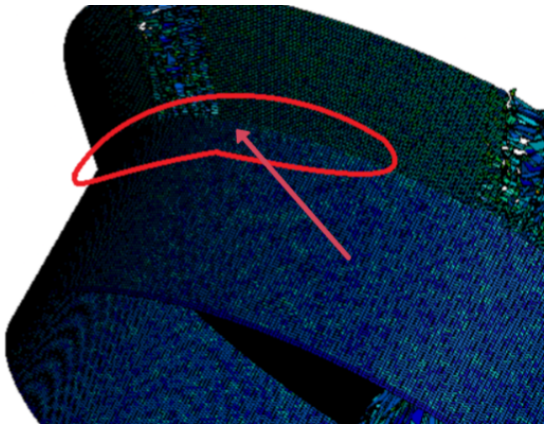


Figure 5.7: Sharp corner between deformed and undeformed state

Moreover, plots for resultant forces and drill torque were achieved. It is worth to

mention that X and Y torque were very small, therefore they were assumed to be zero. Torque values based on FEM results were not crossing values of 50 [Nmm].

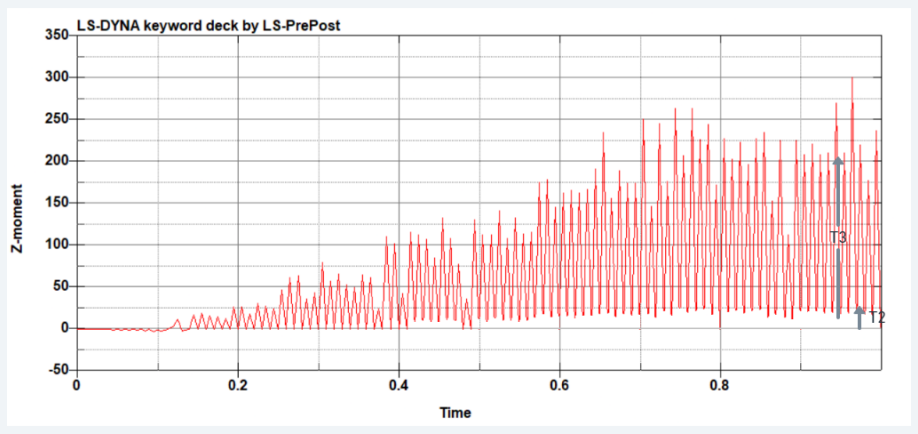


Figure 5.8: Torque of the drill in [Nmm] versus time in seconds.

Based on already existing models and research [32] it is possible to distinguish two torque types within achieved resultant torque from simulation while using shell elements.

Three distinguished torque types are appearing during the procedure of drilling/chamfering of pipe.

The first one, which is not appearing in a plot in Figure 5.8 is described as **cutting torque** and can be described by Equation 5.2 in exponential form. Torque is associated with resistance torque appearing during the cutting of material and normally has a steady uniform value [32].

$$T_1 = C \cdot f^{(1-\alpha)} \cdot D^{(2-\alpha)} \tag{5.2}$$

The coefficient C can be determined through calibration tests, while the material factor α can be obtained from data available in handbooks or reference sources. **Cutting torque** is not appearing in the study due to the fact that in the simulation there is no chip formation.

The second component of torque, which can be assumed as part of the simulation is **Chip evacuation torque**. In this case, this small value of torque symbolizes the plastic deformation of the pipe and its constant cone shape forming from the pipe. **Evacuation torque** can be approximated by the Equation 5.3:

$$T_2 = A \cdot (e^{A \cdot z} - 1) \tag{5.3}$$

A is the parameter, which can be determined by experiments or calibration. Coefficient z is denoted as the depth of cut.

The third component of torque appearing in Figure 5.8 is called **Stick-Slip** and can be described as cumulative resistant torque, which is appearing during especially

deep drilling. According to existing literature [32] the way to approximate and model torque is to use statistical techniques such as the standard Gauss distribution.

In summation, all three components of the torque create the total value of the drilling torque as in Equation 5.4.

$$T = T_1 + T_2 + T_3 \tag{5.4}$$

Based on Plot 5.8 it might be read that approximated value of total torque differs with time, however, it can be approximated by the exclusion of some of the peaks that its value is between 190 - 220 [Nmm].

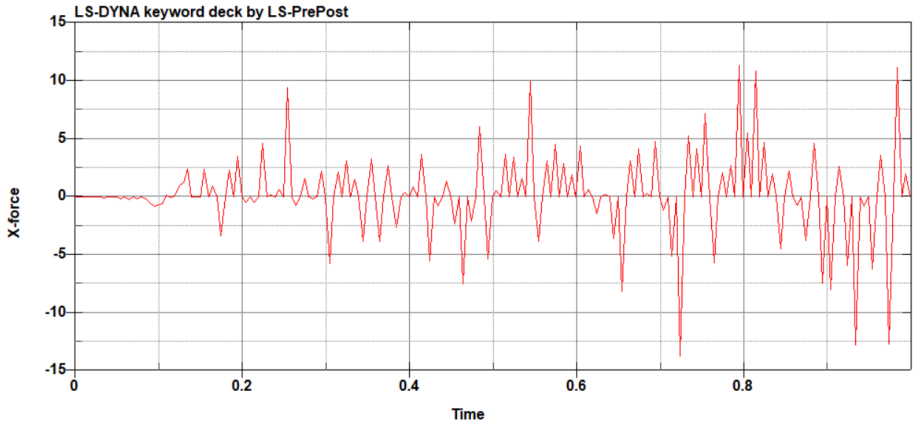


Figure 5.9: Resultant cutting force in Newtons versus time in seconds

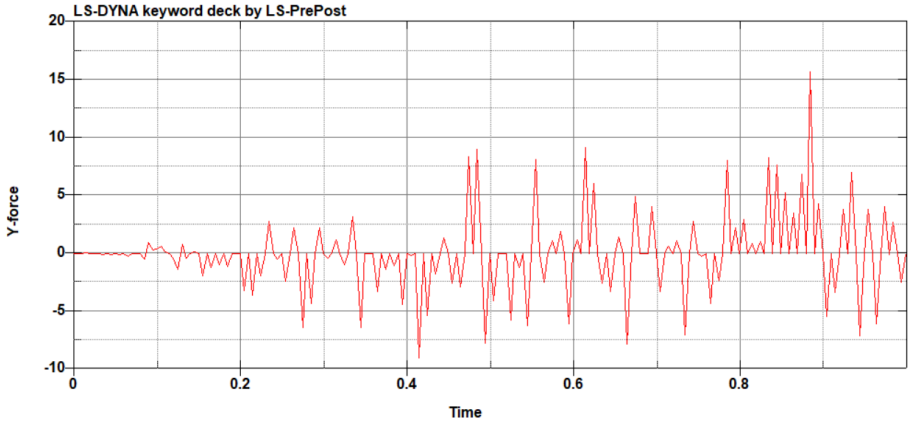


Figure 5.10: Resultant radial force in Newtons versus time in seconds.

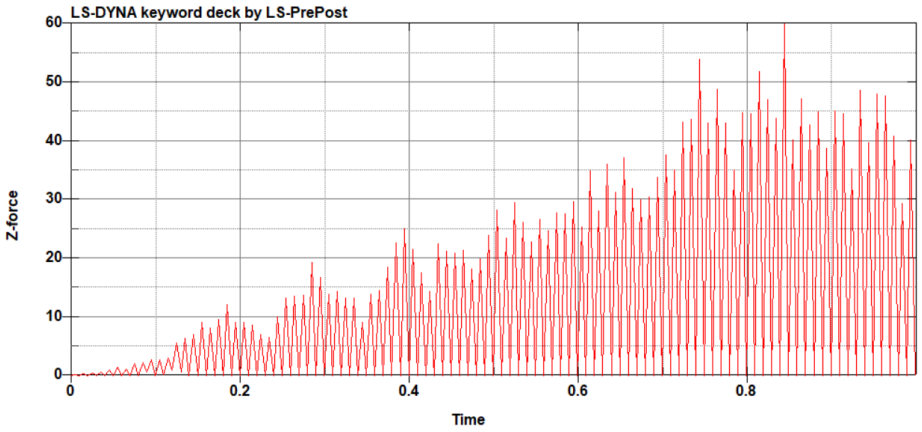


Figure 5.11: Resultant thrust force in Newtons versus time in seconds.

Determination of the direction of forces was based on previous studies [22]. Moreover, each of them was received from simulation and they are represented for each component of force in Figures 5.9, 5.10, and 5.11.

A correlation between thrust force (Z force) and drilling torque might be spotted due to their similarity. While torque is increasing at the same time also thrust force increases. This phenomenon can be explained due to the increasing effort that the drill is subjected to while getting deeper and deeper with chamfering. Resistance of plastically deformed material counteracts against drill tool creating force. Thrust force reaches its peaks at around 50 [N], however, most of the time it is in the range between 30-40 [N].

While looking at Plots of cutting (Figure 5.9) and radial forces (Figure 5.10) it can be noticed that forces have oscillating behavior and according to existing studies [33] they might be explained as the action of the workpiece material on the drill head. Those forces in most cases are the main reason for the vibrations of the drill head, and the wear of it, as well if they are too big, which can happen from too fast speed and too high a feed rate they can cause the drill tool to fail. However, the simulation showed especially in the case of radial force that its approximated value stays around 3-5 [N] and cutting force peaks at 13 [N], however generally its range is between -5 to 5 [N].

5.1.5 Simulation with SPH

The previously mentioned technique of using Smoothed-Particle Hydrodynamics for modeling material was used as an additional attempt in modeling and with better improvement and adjustment is capable of providing a more robust model for chamfering than in the case of using shell elements for the pipe. Due to the much higher instability that was appearing during the faster rotational speed of the drill, the speed was reduced from 3000 to 100 [RPM] for 0.2 seconds of simulation. The simulation was made to provide an answer to the question of a model created based on meshless techniques could be developed in the future and how output resultant forces and drilling torque are similar to the model with shell. Based on existing studies [10] it

was possible to adjust the parameters of the simulation and control its stability of it since SPH can create multiple problems such as viscosity or contact problems when it is not precisely set up. To set up those parameters perfectly it is required to conduct multiple simulation attempts until stability is achieved and results seem adequate as well as the simulation is not terminated due to calculation problems.

Boundary conditions in simulation with SPH elements stayed the same as was previously discussed with the shell method. The bottom of the pipe was fixed and the drill was subjected to movement. The main difference between simulations appeared in the contact form and with control settings. Firstly, the SPH pipe and drill could not be modeled using `Automatic Surface to Surface contact` because SPH particles are modeled as nodes, therefore contact form had to be replaced by `Automatic Nodes to Surface`.

Moreover, one of the most crucial parameters, to adjust the simulation, was `Control Bulk Viscosity`, which provided a much more stable simulation and had the tendency to remove the issue of "exploding" SPH pipe after first contact. According to the author of previous studies [10] the higher value within the range of 0.06 - 1 is desirable to provide robustness, however, the best results and high stability were achieved after multiple tries with Q1=0.5 and Q2=1e-8.

To control general settings of simulation and contact it is appropriate to incorporate settings such as `Control SPH` and `Memory`. The first keyword provides stability in the matter of searching for contact interactions between SPH particles and memory provides either dynamic or static memory allocation for simulation.

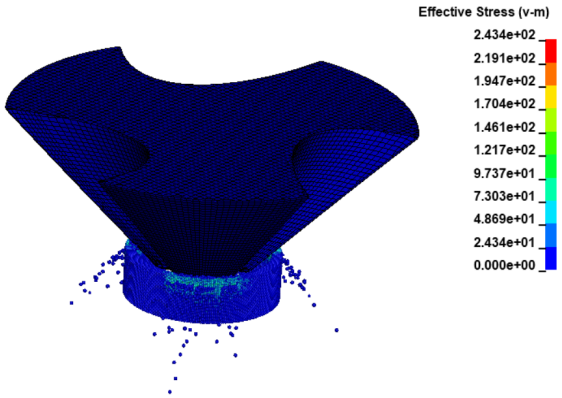


Figure 5.12: Effective stress at initial contact

After initial contact and the first rotational movement of the drill it was noticed that particles tended to be compressed in the outer direction of the pipe as well as to be more subjected to erosion, which creates different results than what appeared in shell simulation as well in real case scenario, see Figure 5.12.

Nevertheless, output resultant forces and resultant drilling torque provided similar values to those that appeared during the shell simulation.

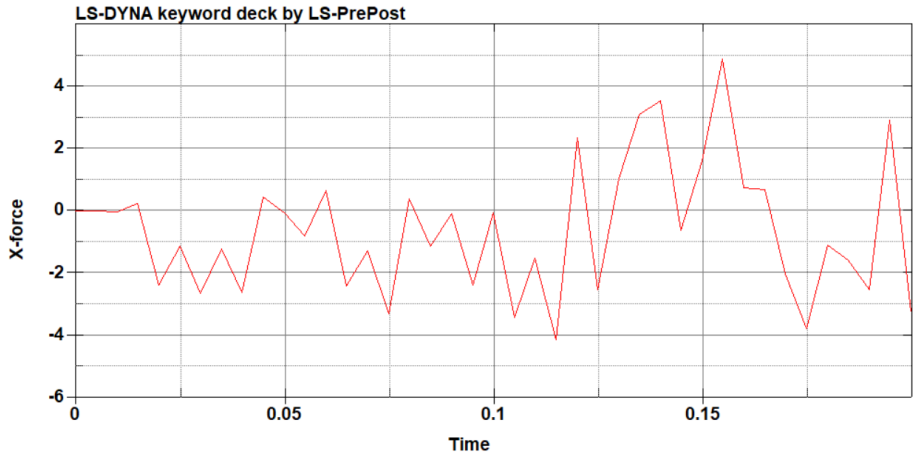


Figure 5.13: Resultant cutting force in Newtons versus time in seconds from SPH.

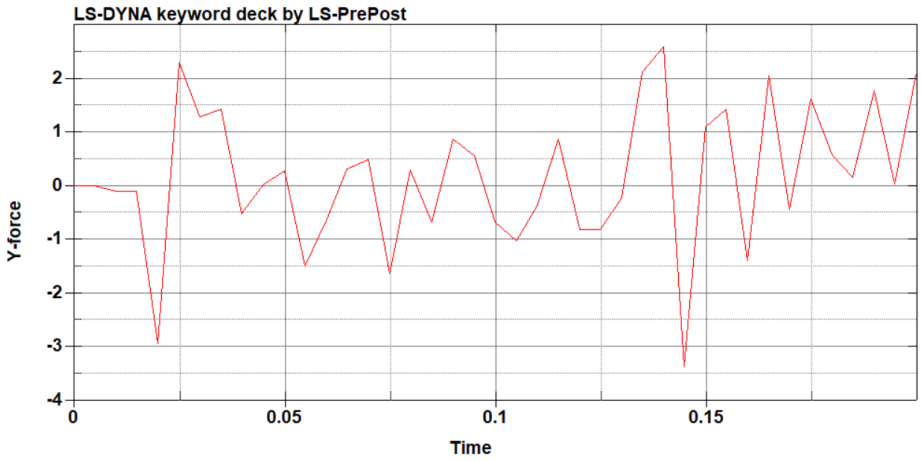


Figure 5.14: Resultant radial force in Newtons versus time in seconds from SPH.

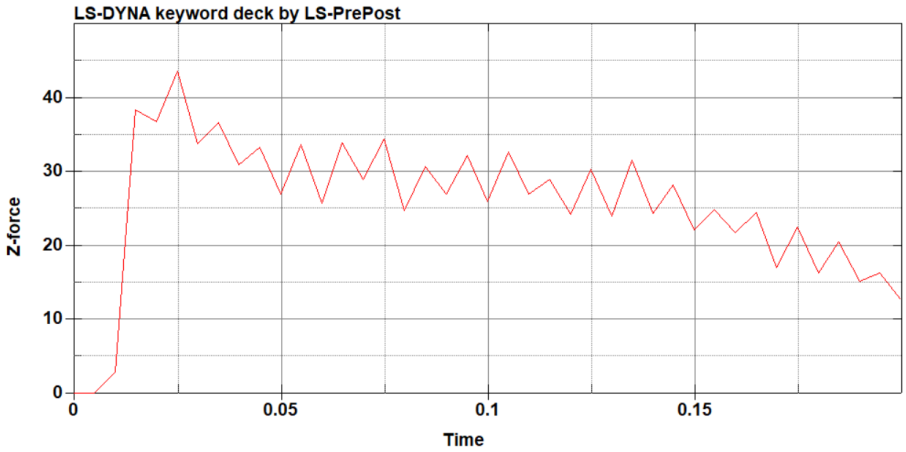


Figure 5.15: Resultant thrust force in Newtons versus time in seconds from SPH.

Looking at plots of resultant forces on Figures 5.13, 5.14 and 5.15 from SPH simulation and comparing them to results from simulation from the shell at Figures 5.9, 5.10 and 5.11, it is easy to find out that resulting values of achieved forces and torques are very close to each other. In the case of SPH simulation forces of X and Y components were smaller, however, this can come from the fact that the RPM of the drill was decreased for stability purposes, and therefore cutting and radial forces from chamfering are lower. In the case of SPH simulation, both are between -2 and 2 Newtons. However, since the feed rate is not changed for the simulation thrust force of both simulations gives similar output, even though force in SPH tends to produce a peak value reaching 40 [N] at the first contact and then slowly decreases with time. This can come from the fact that SPH simulation despite multiple changes in bulk viscosity and contact settings provided much more erosion effect of chips than it is in the case of shell simulation or even real case scenarios. This might come from the fact that several particles for the pipe should be increased to provide smoother chip formation, as well additional keyword settings in simulation should be adjusted. Despite that, the estimated value of 30 [N] of thrust force is very close to the results achieved with the shell body.

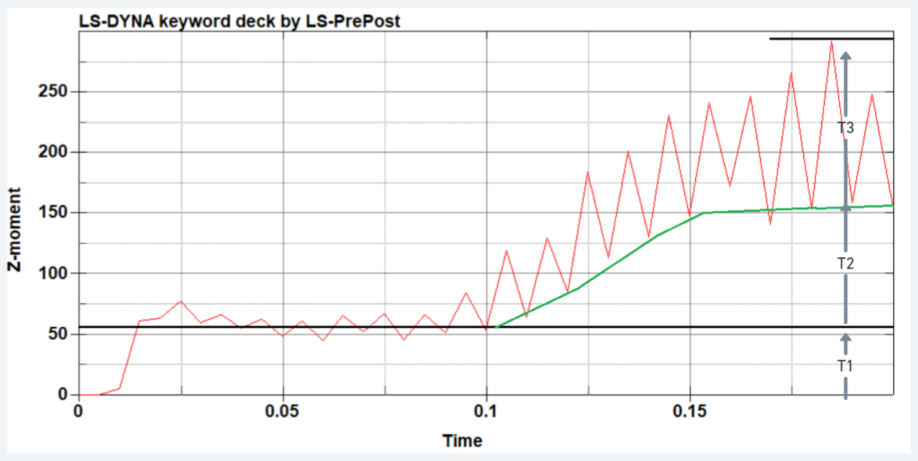


Figure 5.16: Resultant drilling torque in N.mm versus time in seconds from SPH.

While looking at produced drilling moment by simulation using SPH elements at Figure 5.16 it can be noticed that there is a significant difference between curve shapes created by the shell (Figure 5.8) and SPH simulations. The shape of the curve has a very similar output to drilling experimental torques achieved in different studies [32]. Cutting torque can be distinguished based on the stable constant value that appears at the beginning of the procedure, as it is shown in Figure 5.16. Torque is appearing since there is chip formation in the simulation. Moreover, the achieved approximated value of torque from SPH is very close to the drilling torque, which was achieved by simulation containing shell body, which is around 200-220 [Nmm].

The similarity in results can bring the conclusion that achieved values while matching used methods can provide stable tools in order to provide results without experimental setup. Moreover, an attempt to conduct simulation based on SPH elements can provide an idea that higher power resources and computational power could provide very unique and precise results by using the mesh-less technique in chamfering process and give much more adequate results and better performance than standard methods of using rectangular or triangular mesh elements.

5.2 Theoretical model and experimental study

A theoretical model for defining forces and torques appearing during chamfering was established based on a previous study [22]. Based on it, the equation for the definition of cutting force (Equation 5.5) was setup:

$$F_s = K_s \cdot A \tag{5.5}$$

Where K_s is defined as the specific cutting resistance of the material, and normally is defined experimentally as a constant. Parameter A is a chip cross-sectional area.

As well as for drilling torque in Equation 5.6:

$$M_s = F_s \cdot \frac{D_1}{2} \tag{5.6}$$

Where D_1 is specified as the inner diameter of the pipe.

Both approximations of cutting force and torque could be calculated based on the experimental constant for the specific cutting resistance of the material. However, the lack of existing literature and provided data about the value of constant for PLA provided limitations, which could be solved experimentally.

One way to define constant as well as determine the value of cutting force is to establish either cutting force or drilling torque based on values of torques in joints of UR5 after the start of the chamfering procedure. Due to the equipment and ability to conduct the test, reading for each of the joints were collected during the procedure to define properties and parameters for real chamfering situations and compare them to results achieved by FEM simulations.

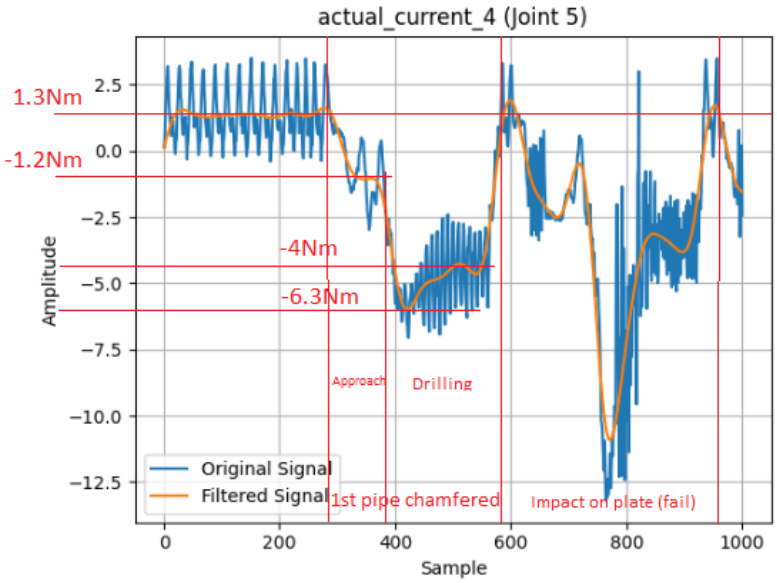


Figure 5.17: Registered torque in joint five of UR [Torque in Nm]

It might be noticed that measurements in Figure 5.17 are very noisy due to vibrations appearing from chamfering, which is why it is hard to determine the approximated value of drilling torque. Additional filtering with low pass Butterworth filter gave approximated values. Based on the plot at Figure 5.17 of Torque in [Nm] provided by measurement on joint number five of UR5e robot it is possible to determine approximated values of thrust force as well as radial and cutting forces. There are three distinguishable sections in the plot. First, there is a steady state at around 1.3 [Nm], where the joint is in the initial position without conducting any movement as well as not being impacted by any external force. Afterward, it can be noticed that the value of torque is increasing in a negative direction till -1.2 [Nm]. That section represents the approach to the first pipe to be chamfered as well as forces that can appear

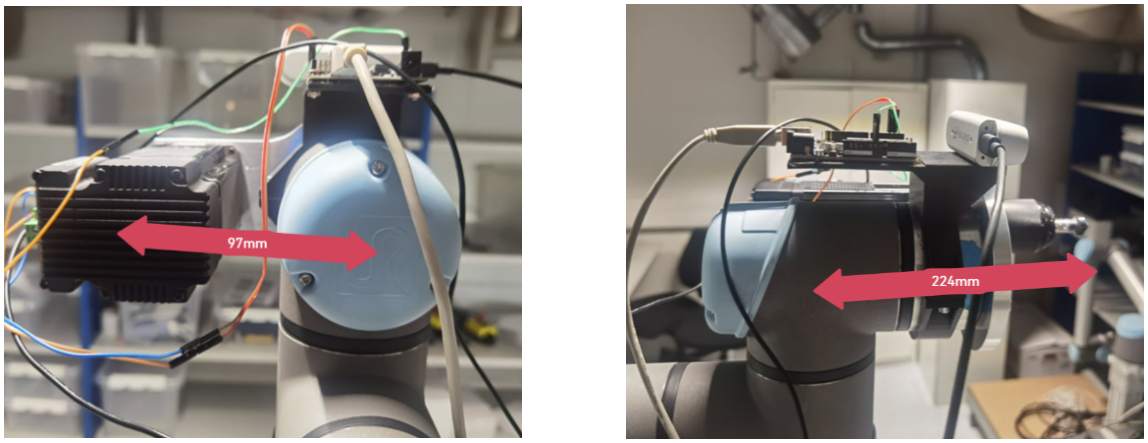
from idle movements of the drill, nevertheless at this point drill is not in contact with the pipe yet. When there is first contact with the pipe for chamfering it can be noticed that there is an instant increase in the appearing moment (from -1.2 [Nm] till -6.3 [Nm]). At this point, it might be assumed that the thrust force from drilling is appearing and it starts to produce a moment on Joint number five. Nonetheless, it can be only approximated at this point that the main and most significant force appearing at the first contact and cut is thrust force, there are still other components of forces including radial and cutting force that might impact results. Although this concern it might be assumed that cutting and radial forces are close to zero due to their oscillating nature as was proved by finite element simulations.

After first contact and progress with chamfering into the pipe, it can be noticed that there is higher fluctuation in the original signal, which oscillates with increasing magnitude until it reaches a steady state value of it. At this point, it may be concluded that appearing fluctuations at the moment can be originated from radial and cutting forces, which are increasing with time.

The filtered curve is showing also a phenomenon, which appeared on the plot of thrust force from SPH simulation (Figure 5.15), where thrust force tended to drop in value with time. The same happens at the measured torque sections of chamfering in the plot of Figure 5.17, which may indicate that the sectional filtered curve represents values of mainly thrust force.

Having analyzed previously described parts of plot in the Figure 5.17, it was possible to define roughly estimated values of thrust force and resultant force of radial and cutting ones.

The moment change between the first approach of pipe and initial drilling is equal to 4.1 [Nm]. To define the value of thrust force it is mandatory to establish the distance between the axis of force and the axis of joint number five, which after the measurement is equal to 0.097 [m] (Figure 5.18a), therefore from the simple relationship, were moment is a product of force and distance gives the approximated value of thrust force to be around 42.27 [N], which almost perfectly follows resultant peak forces achieved from finite elements simulations, especially peak value of thrust force from SPH simulation, see Figure 5.15.



a) b)

Figure 5.18: Offsets from drill position to the center of joint five

Moreover by finding the approximated magnitude of oscillation after initiations of the procedure of chamfering it is possible to define the approximated value of the resultant force of radial and cutting forces. It is almost impossible to distinguish separate values of forces for both radial and cutting ones since they are constantly changing due to the rotation of the drill and cutting edges. Nevertheless, the peak value of the resultant force might be defined, and based on previously mentioned forces it might be roughly estimated.

Approximated oscillating magnitude is equal to 2.3 [Nm] based on differences of values in the plot in Figure 5.17. Having distance to the center of joint five from the tip of the tool (Figure 5.18b), which is equal to 0.224 [m], it is possible to establish the magnitude of force based on the moment and distance relationship. The approximated value provided the resultant force for radial and cutting force to be around 10.28 [N].

If considering that no other forces or moments are acting on the joint, as well as components of radial and cutting force are equal, the Pythagoras relationship provides value for each radial and cutting force to be around 7.27 [N], which provides similar results to those computed by Finite Element Simulation, where pipe body was constructed as shell, see Figures 5.9 and 5.10.

Based on provided results as well as a linear formulation for cutting force based on Equation 5.5 it was possible to define the approximated value of specific cutting resistance for PLA material. Knowing the area of the cut chip, which can be defined based on Equation 5.7:

$$A = \frac{f}{2} \cdot (\frac{D_2}{2} - \frac{D_1}{2}) \tag{5.7}$$

Where f is the feed rate, D_1 value of the inner diameter of the pipe, and D_2 of the outer diameter of the pipe.

Provided data about pipe diameters [Vestas Aircoil] to be respectively D_1=8.8 [mm]

and D_2=10 [mm], as well as established feed rate of 0.15 $[\frac{mm}{rev}]$ made it possible to define the value of the area of the cut chip.

Established from experimental measurement and confirmed by the simulation approach, the value of cutting force, as well as the defined area of the cut chip, provided the approximated value of specific cutting resistance to be 161.56 [MPa].

It might be noticed that Figure 5.17 describes a section called the impact of the plate with fail. Moments appearing in this section describe the failed chamfering process, which appears due to misalignment with coordinates for chamfering and drilling head, as well as a shift in the position of the model of the cooler, which was not fixed to the table with the robust connection. That is why, instead of chamfering pipe drill head hit the plate of the cooler causing impact.

5.3 Summary

Based on provided techniques using explicit dynamic simulations with the usage of mesh or meshless methods, as well as a simple linear theoretical model, which can be established based on experiments it is possible to define properties of chamfering of pipes for multiple materials and use those techniques to define resultant forces and drilling torques based on variable drill speed and feed rate.

6 | Optimization of UR5e Joint Configuration

The UR5e robot has six revolute joints, each of which can rotate freely. It gives the robot a large number of possible configurations. However, not all configurations are equally suitable for every task. In the case of pipe chamfering, the robot should move to each pipe and perform the chamfering operation. The optimal configuration for this task is the one that minimizes torques in the joints and ensures the best distribution of forces during the chamfering process. Given that the robot performs chamfering for a number of pipes, which means that the configuration is not always the same. It depends on the position of the pipes, which are detected by a camera and can vary due to inaccuracies in the camera and the mobile robot MiR200 positioning the UR5e, so it is almost impossible to calculate the optimal configuration for each pipe, without significantly slowing down the work of the robot. For this reason, the approach was chosen to place the robot in an ideal configuration before starting the chamfering process, with calculated forces that occur in the chamfering process, based on the chamfering study explained in Chapter 5. Although it is not an ideal solution, the robot should be kept to some degree of minimized torques within the joints, because when the chamfering procedure is performed, the robot moves linearly towards the pipes and performs the procedure, so the first 3 joints that can handle higher torques are mostly working.

6.1 Optimization Approach

The optimization approach used in this work is based on the Robotics System Toolbox in MATLAB. The toolbox provides a comprehensive set of tools for modeling, simulating, and analyzing kinematics, dynamics, and control of robots [34]. The optimization algorithm used is the `fmincon` function in MATLAB, which finds the minimum of a constrained nonlinear multivariable function. The algorithm takes into account the weight of the drill, the torques, and forces that occur during chamfering, and the position and orientation of the end-effector at the start of the chamfering procedure.

6.2 Optimization Process

The optimization process involves several steps:

1. Define the robot model using the Denavit-Hartenberg parameters of the UR5e robot. The model includes the mass, center of mass, and inertia tensor of each link. The drill is modeled as an additional link attached to the end effector of the robot.
2. Define the desired end-effector pose. This is the pose that the end effector should have when it is set before performing the process of chamfering of pipes.

3. Define the cost function for the optimization. The cost function in an optimization problem is a mathematical function that the algorithm tries to minimize or maximize. In this case, the cost function is a weighted sum of the pose error, the difference in torques between the initial and current configurations, and the last joint angle. The weights are chosen based on the importance of each term in the cost function.
4. Perform the optimization from multiple starting points. This is done to avoid getting stuck in local minima and to find the global minimum of the cost function. The starting points are chosen randomly within the joint limits of the robot.
5. For each starting point, perform the optimization using the `fmincon` function. The function returns the joint configuration that minimizes the cost function.
6. Compare the cost of the current solution with the best cost found so far. If the current cost is lower, update the best cost and the best solution.
7. After all starting points have been tried, the best solution is chosen as the optimal configuration.

6.3 Mathematical Formulation of the Optimization Problem

The complete code for the optimization of UR5e joint configuration based on chamfering study can be found in Appendix G. To give a better insight, the optimization problem can be formulated as follows:

$$\min_{q} \quad w_{\text{pose}} \cdot \text{pose\_cost\_function}(q) + w_{\text{torque}} \cdot \|\tau_{\text{initial}} - \tau_{\text{current}}(q)\| + w_{\text{joint6}} \cdot |q_6| \quad (6.1)$$

subject to

$$q_{\min} \leq q \leq q_{\max} \quad (6.2)$$

where q is the vector of joint angles, w_{pose}, w_{torque}, and w_{joint6} are the weights for the pose error, the torque difference, and the last joint angle, respectively, τ_{initial} is the vector of initial joint torques, $\tau_{\text{current}}(q)$ is the vector of current joint torques, $q_{\min}$ and $q_{\max}$ are the lower and upper joint limits, respectively.

The pose error cost function is defined as:

$$\text{pose\_cost\_function}(q) = \|\mathbf{p}_{\text{desired}} - \mathbf{p}_{\text{current}}(q)\| + \|\mathbf{r}_{\text{desired}} - \mathbf{r}_{\text{current}}(q)\| \quad (6.3)$$

where $\mathbf{p}_{\text{desired}}$ and $\mathbf{r}_{\text{desired}}$ are the desired position and orientation of the end-effector, respectively, and $\mathbf{p}_{\text{current}}(q)$ and $\mathbf{r}_{\text{current}}(q)$ are the current position and orientation of the end-effector, respectively.

The current joint torques are computed as:

$$\tau_{\text{current}}(q) = \tau_{\text{gravity}}(q) + J_{\text{offset}}(q)^T \cdot \mathbf{f}_{\text{drill}} \tag{6.4}$$

where $\tau_{\text{gravity}}(q)$ is the vector of joint torques due to gravity, $J_{\text{offset}}(q)$ is the Jacobian matrix at the offset position, because of not having a drill in the center of the end-effector, and $\mathbf{f}_{\text{drill}}$ is the vector of drilling forces and torques.

The optimization problem is solved using the `fmincon` function in MATLAB, which finds the minimum of a constrained nonlinear multivariable function. In this case, the function to be minimized is the cost function, and the constraints are the joint limits of the UR5e. The `fmincon` function uses various optimization algorithms such as interior-point, trust-region-reflective, and sequential quadratic programming. The specific algorithm used can be selected by setting the 'Algorithm' option in the options parameter of the `fmincon` function [35]. The function is called multiple times from different random starting points to avoid getting stuck in local minima and to find the global minimum of the cost function.

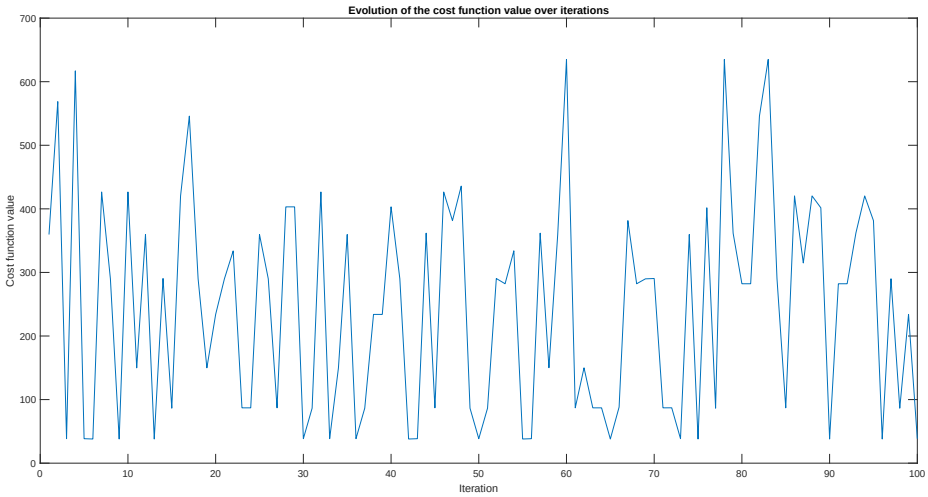


Figure 6.1: Evolution of the cost function value over iterations.

Figure 6.1 shows the evolution of the cost function value over iterations for desired end-effector pose. The plot of the cost function value over iterations shows a fluctuating curve. The optimization algorithm tries to find the minimum value of the cost function, but it also needs to explore the solution space to avoid getting stuck in local minima. The final value of the cost function and the appropriate configuration will depend on the specific details of the optimization problem and the settings of the optimization algorithm.

6.4 Optimization Results and Validation

The optimization process was confirmed in a step-by-step manner, starting with the UR5e robot in isolation, then adding only the drill, and finally including the forces and torques that occur during the chamfering process. In this way, it was easier to confirm the validity of optimization and fault finding in the final model. For the first two methods, desired pose end-effectors were selected, optimization was done and then the torque values obtained in these configurations were compared against the values read on the physical robot. While for the last part, with the inclusion of forces and torques, some configurations were entered that were read during chamfering and compared with the calculation of torques in the optimization program, in order to confirm the validity of the calculation of torques because setting the desired position of the end effector was not possible because chamfering is performed for all pipes.

6.4.1 Optimization of UR5e Alone

Initially, the optimization was performed exclusively on the UR5e robot without any additions on the end-effector, or external forces and torques. The robot was placed in several specific configurations, based on desired end-effector poses, and joint torques were calculated using the Robotics System Toolbox in MATLAB. Those configurations are [°]:

- Configuration 1: $\begin{bmatrix}0.540 & -144.25 & -0.110 & -36.360 & 0.00 & 0.00\end{bmatrix}$
- Configuration 2: $\begin{bmatrix}-150.430 & -34.51 & 103.417 & -243.734 & -87.373 & -26.17\end{bmatrix}$
- Configuration 3: $\begin{bmatrix}-106.251 & -34.906 & 103.436 & -243.704 & -87.321 & 4.921\end{bmatrix}$

These calculated values were then compared to the actual readings of the UR5e robot in those configurations. The results are shown in Table 6.1. It can be noticed that results are not identical, but that was not to be expected because many aspects are at play when calculating the torque, from the exact definition of masses, and moments of inertia to noise in the readings. But they showed a close match between the calculated and actual values, thus validating the optimization process for the UR5e robot in isolation.

Joint	**Configuration 1**		**Configuration 2**		**Configuration 3**	
	MATLAB	**Physical**	**MATLAB**	**Physical**	**MATLAB**	**Physical**
1	0.0000	-2.4997	0.0000	-1.2979	0.0000	1.6537
2	-48.5403	49.0214	41.4038	-42.3175	41.3446	-41.5724
3	-15.0083	11.3625	7.3602	-5.4302	7.4609	-4.8965
4	-0.0229	1.4078	0.7223	-2.6306	0.7118	-1.5285
5	0.0070	0.4925	-0.0023	-1.1163	-0.0022	1.9569
6	0.0000	0.4342	0.0000	0.4997	0.0000	-0.1308

Table 6.1: Comparison of torques [Nm] for different configurations in UR5e joints

6.4.2 Inclusion of the Drill

Then a drill was added to the robot model and the optimization process was repeated, in the same way as for isolated UR5e. Configurations care following [°]:

- Configuration 1: $\begin{bmatrix}-117.401 & -24.898 & 102.90 & -257.90 & -63.201 & -0.005\end{bmatrix}$
- Configuration 2: $\begin{bmatrix}-136.615 & -69.528 & 78.075 & -209.089 & -55.685 & 24.259\end{bmatrix}$
- Configuration 3: $\begin{bmatrix}258.6500 & -61.9600 & 127.1000 & -245.510 & -77.470 & 2.77\end{bmatrix}$

The calculated torques, including the effect of the drill, were compared with the actual readings of the UR5e robot equipped with the drill. As shown in the Table 6.2, it was again determined that the calculations are not identical, largely because the moments of inertia and the position of the center of mass on the drill were estimated based on the Solidworks model, which gives a certain error. But again the values are quite close to the actual readings, thus confirming the optimization procedure for the UR5e drill robot.

Joint	**Configuration 1**		**Configuration 2**		**Configuration 3**	
	MATLAB	**Physical**	**MATLAB**	**Physical**	**MATLAB**	**Physical**
1	0.0000	-3.5737	0.0000	-1.2284	0.0000	-0.0642
2	-52.4423	-49.2058	-43.4051	-47.7318	-36.5708	-32.3587
3	-6.2673	-2.5443	-25.6011	-28.3099	-12.6402	-13.4999
4	-0.5944	0.2017	1.3842	1.0686	-1.1681	-2.9879
5	0.0045	-0.7208	-0.8665	-1.8432	-0.0150	-1.0092
6	2.0012	2.3687	1.8840	1.2787	2.0019	3.0508

Table 6.2: Comparison of torques [Nm] for different configurations with included drill

6.5 Inclusion of the Torques and Forces occurred during Chamfering Process

Finally, the chamfering process is included in the optimization process. The calculated torques, now including the forces and torques due to chamfering, are compared to the actual readings during the chamfering process. Although it is very difficult to compare the values during the chamfering process, an attempt was made to obtain readings from the physical robot torques and joint states during chamfering, and it was tried to obtain values from the optimization code for the joint configurations. Based on the Chamfering Study (Chapter 5), the mean values of the forces and torques that occur during the chamfering process were selected:

$$F_x = 7.27\,[\text{N}] \qquad \tau_x = 0.05\,[\text{Nm}]$$
$$F_y = 7.27\,[\text{N}] \qquad \tau_y = 0.05\,[\text{Nm}]$$
$$F_z = 42.27\,[\text{N}] \qquad \tau_z = 0.22\,[\text{Nm}]$$

Tests were conducted on the configurations used during the chamfering process. Those configurations are [°]:

- Configuration 1: $\begin{bmatrix} -114.274 & -24.278 & 82.173 & -237.784 & -66.260 & 0.064 \end{bmatrix}$
- Configuration 2: $\begin{bmatrix} -110.856 & -32.705 & 87.020 & -234.039 & -69.765 & 0.0581 \end{bmatrix}$
- Configuration 3: $\begin{bmatrix} -120.273 & -30.531 & 81.677 & -230.904 & -60.419 & -0.0306 \end{bmatrix}$

In Table 6.3 values for torques obtained through optimization code and readings from physical robot were shown.

Joint	**Configuration 1**		**Configuration 2**		**Configuration 3**	
	MATLAB	**Physical**	**MATLAB**	**Physical**	**MATLAB**	**Physical**
1	-4.119	2.787	-4.386	18.357	-3.5926	18.4377
2	-89.641	-75.531	-86.723	-55.125	-89.2682	-44.5946
3	-29.682	-28.634	-31.977	-43.454	-33.1699	-40.9283
4	-3.787	-1.024	-4.073	5.497	-3.3381	10.6012
5	-0.660	-0.789	-0.649	-4.540	-0.6472	-11.8122
6	1.793	2.727	1.793	3.361	1.7932	1.2587

Table 6.3: Comparison of torques [Nm] for different configurations during chamfering

The calculated values were not so close to the actual readings, although they were following some kind of distribution trend as in the readings on the physical robot. As mentioned, it is very difficult to imitate the dynamic process of chamfering for only one specific moment, because the optimization code always uses the same mean value of forces and torques based on the chamfering study, therefore large differences are to be expected, even though during the chamfering of the PLA material and do not cause excessive forces. These differences can also be attributed to various factors, such as noise in the readings from physical robots and the difficulty in replicating accurate real-world conditions in the calculations. Despite these differences, the similar match in trend between the calculated and actual values, and based on studies not including forces and torques, could confirm that optimization code is giving realistic results for torques in certain configurations and thus can be used with the optimization part.

In the final phase, the calculation of the optimal configuration for the chamfering process was made. Tests conducted earlier revealed that the optimal position for performing all computer vision detections is at a distance of 300-400 mm from the cooler. Consequently, we evaluated the optimal position and determined the appropriate position of the end-effector relative to the base frame, which is as follows:

$$\begin{aligned} X &= 4.1\,[\text{mm}] & R_x &= 4.794\,[\text{rad}] \\ Y &= 454.58\,[\text{mm}] & R_y &= 0.063\,[\text{rad}] \\ Z &= 18.69\,[\text{m}] & R_z &= 0.135\,[\text{rad}] \end{aligned}$$

where X, Y, and Z represent the position of the end effector in the base frame, and

R_x, R_y, and R_z represent the orientation of the end effector in terms of roll, pitch, and yaw angles respectively.

Based on that position of the end-effector, the optimization code resulted in the following configuration and torques in that configuration:

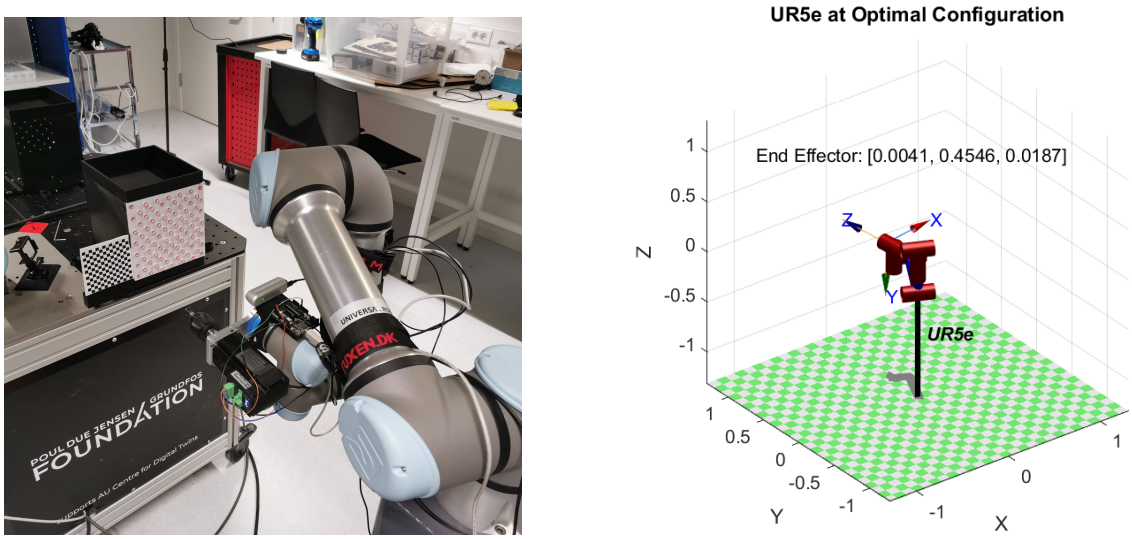
Optimal Configuration [°]:

$$\begin{bmatrix} -114.7259 & -19.2248 & 119.1129 & 78.0818 & -57.7714 & 0.0000 \end{bmatrix}$$

Torques in Optimal Configuration [Nm]:

$$\begin{bmatrix} -3.2147 & -61.7228 & 0.7770 & -2.8313 & -0.9170 & 1.7531 \end{bmatrix}$$

Optimal configuration used for chamfering process can be seen in Figure 6.2:



a) Physical

b) MATLAB model

Figure 6.2: UR5e in optimal configuration

6.6 Summary

This chapter presents a comprehensive study on optimizing the joint configuration of the UR5e robot for the pipe chamfering process. The optimization process aims to minimize the moments in the robot's joints and distribute them evenly during the chamfering process. The challenge lies in the different positions of the pipes and inaccuracies in the positioning of the robot and camera system. The optimization algorithm, implemented using the Robotics System Toolbox in MATLAB and the `fmincon` function, uses a cost function that takes into account the pose error, the torque difference, and the back angle of the joint. Although it is difficult to verify if this is really the optimal configuration for a certain position of the end-effector, it was compared with some other configurations and it really showed lower torques in

the joints. The results of the optimization process are confirmed by comparing the calculated torques with the actual torques measured on the UR5e robot. The chapter ends by acknowledging certain differences between calculated and actual moments due to various factors such as noise in the readings, inaccuracy in the calculation of forces and torques in drilling, not including certain factors in the torque calculations in the code, etc. Finally, the optimal configuration for the chamfering process was calculated.

7 | Computer Vision

7.1 Introduction

The main part of the process incorporates camera vision for pipe recognition and the definition of coordinates for the robot arm to conduct the chamfering procedure of pipes. In this manner, it is possible to align the robot arm in an adequate position. The section explains multiple methods used to define the coordinates of pipes in a cooler while using camera vision to achieve the most accurate and stable results. The initial part of the section explains methods used to orientate the end-effector of the robot arm with the usage of camera calibration.

7.2 Eye-in-hand camera calibration

Hand-eye calibration represents a critical procedure in the realm of robotic applications, with the principal aim of determining the transformation between the robot's end-effector (hand) and an attached camera (eye). This transformation proves integral for tasks such as object manipulation, wherein precise coordination between the robot's movements and the camera's perspective is required.

The mathematical formulation for solving the hand-eye calibration is presented by the AX=XB problem, where A and B are transformation matrices. One solution is proposed by Tsai and Lenz [36] and uses dual quaternions to solve this problem:

$$AX = XB \tag{7.1}$$

In this equation, A is the transformation from the robot base to the robot end-effector, X is the unknown transformation from the robot end-effector to the camera, B is the transformation from the camera to the calibration target, and B is known from vision data.

An important decision in any hand-eye calibration process is the choice between an eye-in-hand configuration and an eye-to-hand configuration. In an eye-in-hand configuration, the camera is affixed to the robot's end effector, whereas in an eye-to-hand configuration, the camera is mounted at a fixed location in the robot's environment and the target is attached to the robot's end effector.

Each configuration carries its advantages and considerations, dictated by the specificities of the task the robot is designed to perform. The eye-in-hand configuration is often preferable in scenarios where the robot is intended to interact with objects in its environment, as it allows the camera to capture images from the robot's perspective, thereby facilitating more precise manipulation.

Given these considerations, the eye-in-hand configuration was selected for the camera placement in this project. This choice was informed by the nature of the tasks the

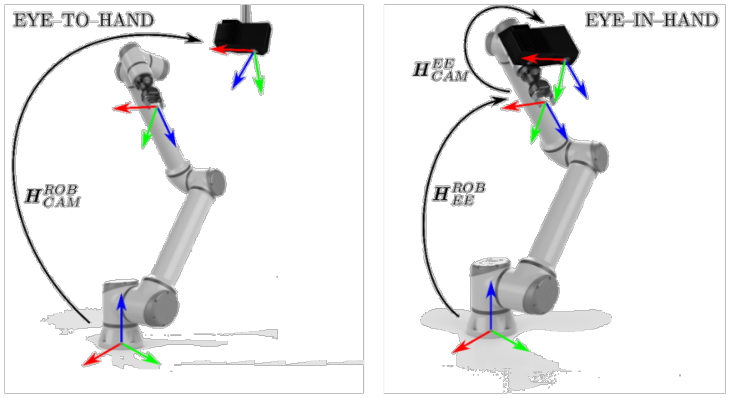


Figure 7.1: Difference between eye-to-hand and eye-in-hand camera calibration [37]

robot is intended to perform.

7.2.1 MoveIt calibration

The process of hand-eye calibration can be conducted via a multitude of methodologies. In the context of this research, the chosen approach was one of the most often employed, which is implemented through the hand-eye calibration tool provided by the MoveIt package. This package offers an array of plugins, alongside a graphical user interface, enabling a streamlined and user-friendly calibration procedure.

The calibration process generally comprises the following steps:

1. Data Collection: The robot is maneuvered through an assortment of poses while concurrently capturing images of a calibration target, typically a checkerboard pattern. The pose of the robot's end effector and the pose of the calibration target, as perceived by the camera, are both recorded.

2. Pose Estimation: The poses of the calibration target in the images are then estimated, which is commonly accomplished by identifying the corners of the checkerboard pattern and utilizing them to compute the target's pose relative to the camera.

3. Calibration Computation: Following the collection of hand poses and eye poses, a calibration algorithm is employed to compute the hand-eye transformation. The computational process involves a nonlinear optimization problem that seeks to minimize the discrepancy between the transformation computed from the hand-and-eye poses and the actual hand-eye transformation.

4. Validation and Refinement: The computed hand-eye transformation is subsequently validated by applying it to the observed hand and eye poses to check for consistency. If necessary, the calibration process can be refined by gathering more data or by tuning the calibration algorithm parameters.

In the context of the MoveIt package, most of these steps are automated, minimizing the manual intervention required from the user and simplifying the calibration

process. The user's involvement is primarily restricted to the initial setup of the calibration environment and the final movement of the robot.

The first requirement is the generation of a calibration target, which is typically printed and measured to determine the size and separation of the markers. The calibration target is subsequently fixed at a specified location within the robot's operational environment. Following this, the user must specify the image and camera info topics within the RViz program. In this project, the required drivers for the Intel RealSense camera were installed in the Robot Operating System (ROS) to facilitate this process.

The next stage involves the selection of the appropriate frames for the calibration. The user is required to specify the sensor frame, object frame, end-effector frame, and robot base frame. It is important to note that the calibration process cannot be initiated until all these frames have been correctly specified. This part of the procedure can also be seen in Figure 7.2, together with a live view from the camera that shows the target that was generated in the previous step and the origin frame found by the software.

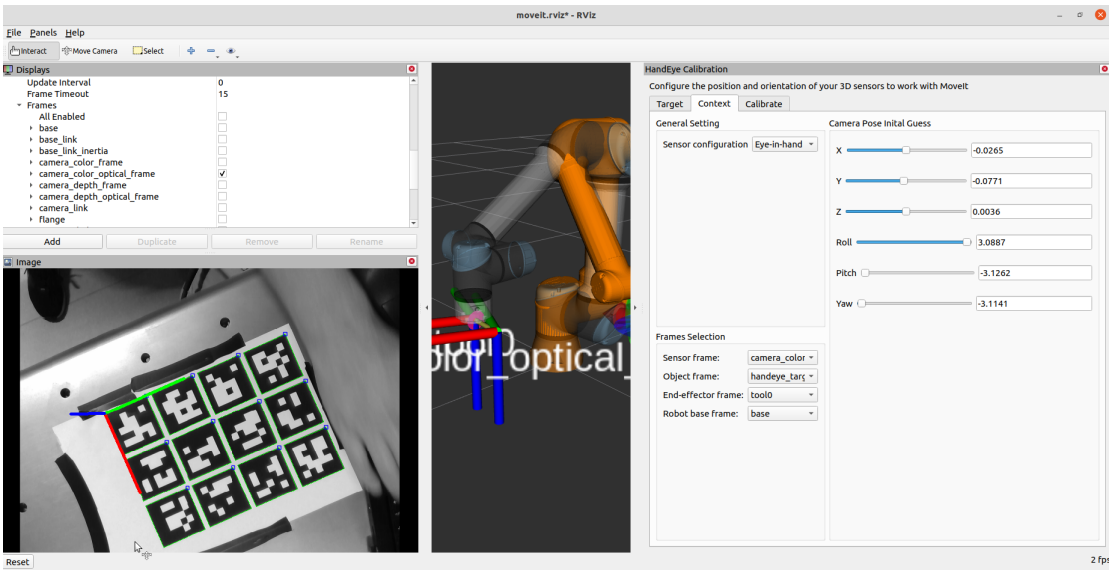


Figure 7.2: MoveIt Calibration package in RViz

Once the above steps have been completed, the robot can be maneuvered through various configurations to collect the necessary calibration data. The robot's joint states and corresponding images of the calibration target are recorded for each pose. After the fifth sample, an initial solution for the hand-eye transformation is computed. However, to obtain a more accurate calibration, it is recommended to collect between 12 to 15 samples. Beyond this range, the improvements in the calibration accuracy are typically marginal and the solution tends to plateau.

The solution of the calibration can be seen in Figure 7.3

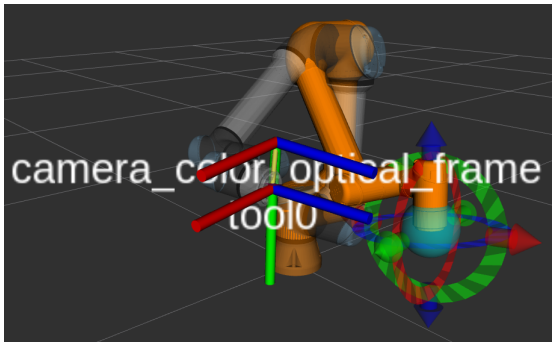


Figure 7.3: Transformation from end-effector to camera's frame

In addition to the above-mentioned steps, the user must also select an appropriate solver for the calibration computation process. The chosen solver is tasked with formulating the nonlinear optimization problem and computing the hand-eye transformation. In the context of this project, the solver implemented is the one presented by Daniilidis in his paper titled "Hand-Eye Calibration Using Dual Quaternions" [38]. This solver is based on the concept of dual quaternions, which provide a compact and efficient representation for spatial transformations, and it has proven to be an effective tool for hand-eye calibration.

7.2.2 Results and analysis

In the presented study, the calibration results were derived from an analysis of multiple tests. An analysis of the camera calibration data in Figure 7.4 showcases a remarkable consistency across each variable. For instance, if the second test yields a high value for the 'X' coordinate, all other variables manifest elevated values for that test. Similarly, a dip is observed in the values of all variables for the fourth test. This consistency may be attributed to the intrinsic nature of the hand-eye calibration process, where changes in one variable are likely to cause shifts in others. Furthermore, the dip observed in the fourth test may be due to an outlier in the data or a momentary anomaly in the calibration process. The final calibration result was determined as the mean of five separate tests. Opting for multiple tests and subsequent averaging serves to enhance the robustness of the calibration process. This approach can effectively mitigate the impact of outliers or irregular fluctuations, thereby ensuring a more reliable and accurate calibration outcome.

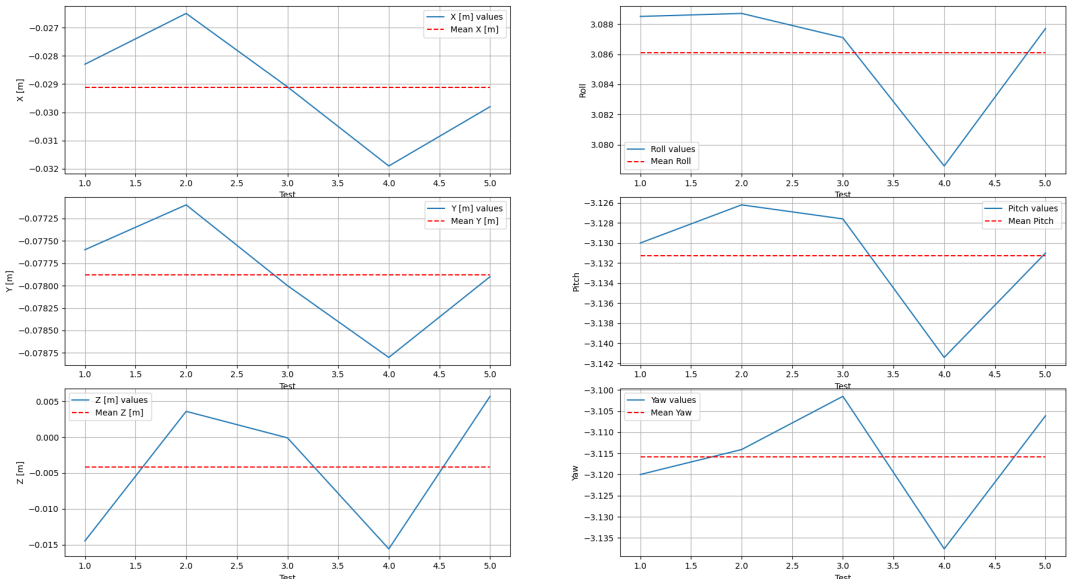


Figure 7.4: Results of hand-eye calibration with mean values

$$T = \frac{1}{n} \sum_{i=1}^{n} T_i \tag{7.2}$$

In the above equation, T represents the final transformation matrix, n is the number of tests, and T_i refers to the transformation matrix obtained from the i^{th} test. Final transformation matrix:

Based on the calculated position and orientation values, the transformation matrix was subsequently computed. The position values are represented by the coordinates X, Y, and Z, and the orientation is defined by the rotation around each axis, denoted as a roll (α), pitch (β), and yaw (γ).

The general form of a transformation matrix for 3D space is given by:

$$T = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix} \tag{7.3}$$

where R is the rotation matrix derived from roll, pitch, and yaw, and $d = [X, Y, Z]^\top$ is the translation vector.

In this particular case, the computed transformation matrix, utilizing the mean values from the calibration process, is:

$$T = \begin{bmatrix} 0.99961536 & -0.02511644 & -0.01175969 & -0.02912 \\ 0.02572843 & 0.998146 & 0.05515989 & -0.07788 \\ 0.01035247 & -0.05544124 & 0.99840828 & -0.00418 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.4}$$

The resulting transformation matrix accurately encapsulates the position and orientation of the camera with respect to the robot's end effector. This transformation matrix is vital to the success of hand-eye calibration, as it allows for accurate coordinate transformations between the camera and robot coordinate systems, thereby enabling precise robotic manipulations based on the camera data.

7.3 End-effector alignment

7.3.1 Camera Calibration

Camera calibration is a very important preparatory step in many computer vision and robotics applications. It serves as a procedure to estimate the internal parameters (focal length, optical center, etc.) and external parameters (rotation and translation vectors) of the camera. Intrinsic parameters refer to the internal characteristics of the camera, while extrinsic parameters describe its position and orientation in the 3D world. These parameters are crucial in determining the exact relationship between the camera's image sensor and the 3D world [39].

Given a 3D point $P_w = (X_w, Y_w, Z_w)^T$ in the world coordinate system, its corresponding point $P_c = (X_c, Y_c, Z_c)^T$ in the camera coordinate system can be determined by:

$$P_c = R(P_w - C) \tag{7.5}$$

where: - R is a 3x3 rotation matrix, - $C = (X_0, Y_0, Z_0)^T$ is the camera center in the world coordinate system.

The camera matrix M is composed of intrinsic and extrinsic parameters and relates the camera coordinates to the pixel coordinates in the image frame. It is typically represented as:

$$M = K[R \;\; t] \tag{7.6}$$

where: - K is the intrinsic parameter matrix, often called the camera matrix, which includes the focal length, the optical center, and the skew coefficient, - R is the rotation matrix, - t is the translation vector.

K can be expressed as:

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{7.7}$$

where: - f_x, f_y are the focal lengths in terms of pixel dimensions, - c_x, c_y are the coordinates of the principal point (optical center), - s is the skew coefficient.

The camera matrix M is used to project the 3D points in the camera coordinate system onto the 2D image plane:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = M \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{7.8}$$

where (u, v) are the pixel coordinates of the projected point and s is an arbitrary scale factor.

The Intel RealSense D435 camera used in this study is a depth-sensing camera known for its exceptional performance in 3D object recognition. Before any operation, this camera is calibrated using a checkerboard, which can be seen in Figure 7.5. This is common practice in camera calibration due to the high checkerboard contrast and regular geometry.

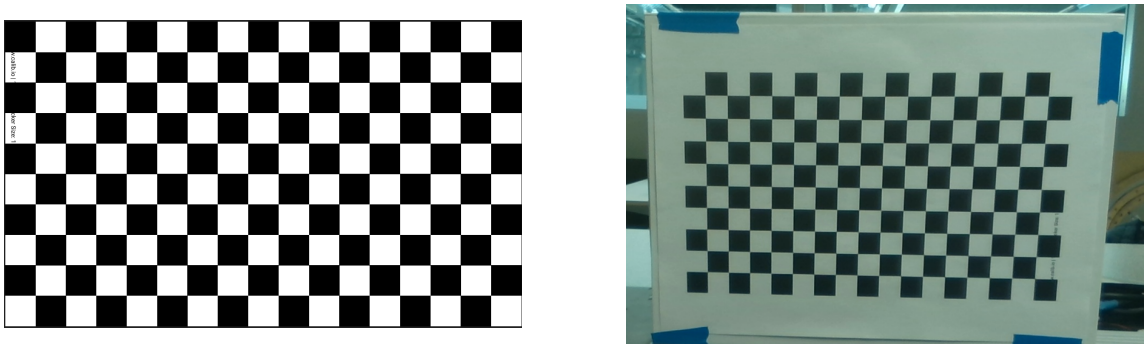


Figure 7.5: An image of the checkerboard used for camera calibration

OpenCV (Open Source Computer Vision) is an open-source computer vision and machine learning software library. It contains more than 2500 optimized algorithms that are comprehensive in computer vision and machine learning. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, extract 3D object models, and produce 3D point clouds from stereo cameras, among other capabilities, making it a very powerful piece of software [40].

The OpenCV library was used in the calibration process. It provides a number of functionalities, one of which includes an application programming interface (API) specifically designed to find the checkerboard corner points in an image, which is essential in our camera calibration process.

The calibration process started by placing a checkerboard in front of the camera and recording a series of images from different angles and under different conditions, shown in Figure 7.6.

Using the captured images, OpenCV's function `findChessboardCorners` was then used to identify the corners of the chessboard [41]. This function takes a grayscale image as input and determines whether a checkerboard is present in the image or not. If there is a checkerboard, the function locates its corners with sub-pixel accuracy, as shown in Figure 7.7.

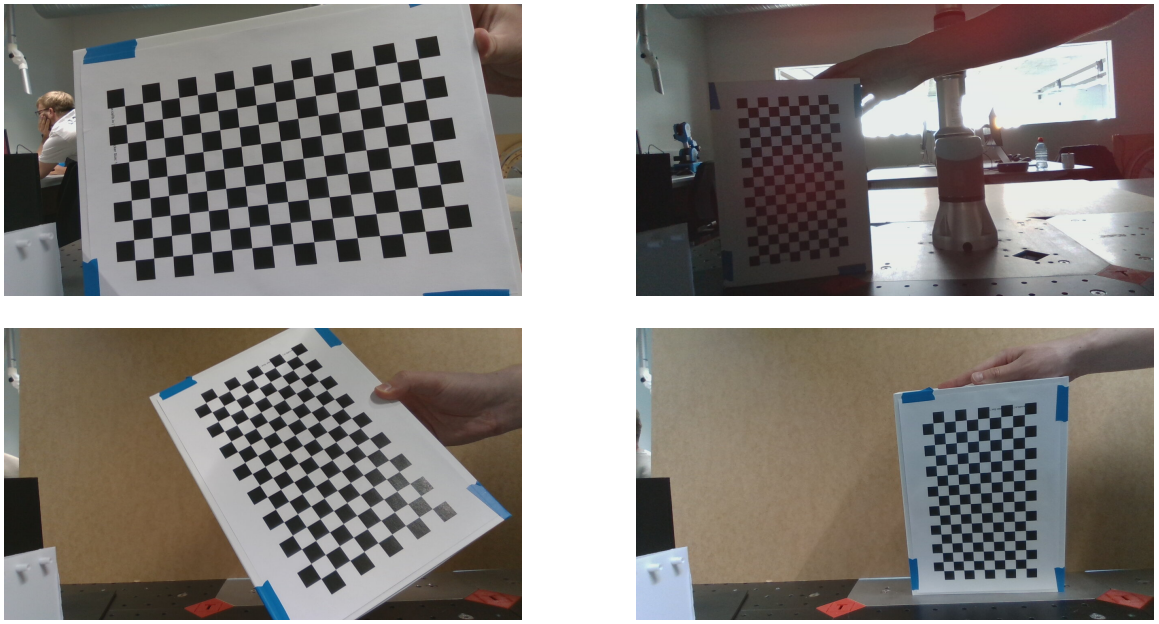


Figure 7.6: A series of images captured from various angles and conditions.

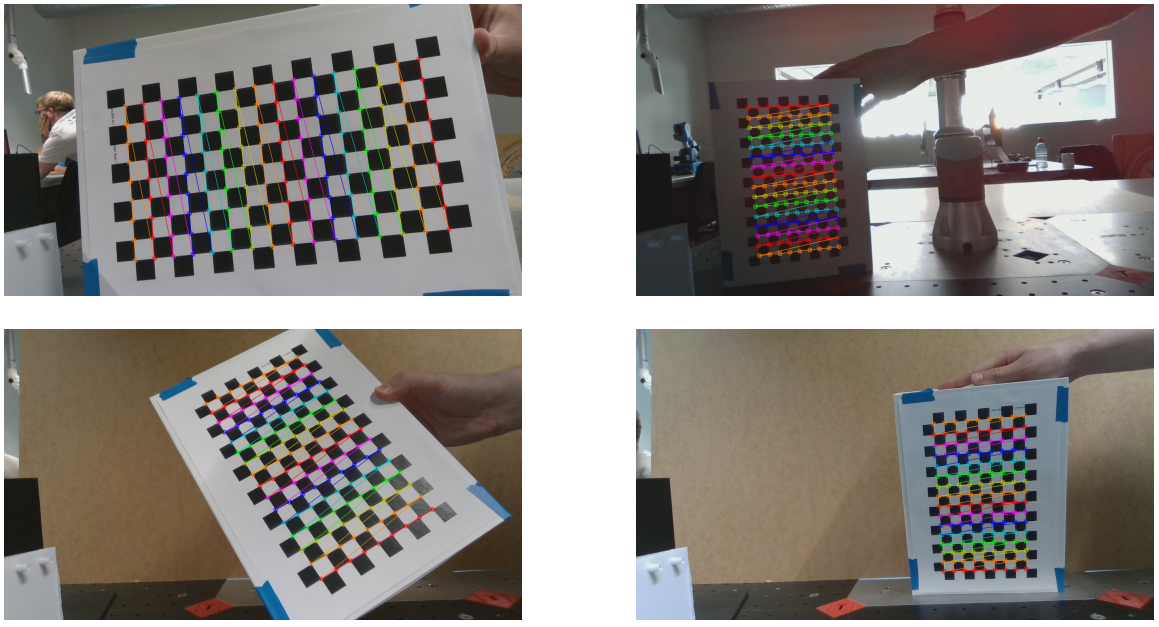


Figure 7.7: An image showing the corners of the squares in the checkerboard identified by the `findChessboardCorners` function.

After the corners were identified, the 3D coordinates of those points (object points) and their corresponding 2D coordinates on the image (image points) were determined. These 3D-2D point pairs served as input to OpenCV's `calibrateCamera` function [41], which estimated internal camera parameters, distortion coefficients, and rotation and translation vectors for each image [42].

This function uses the mathematical model of the pinhole camera, defined by the equation 7.8, to estimate these parameters. Also, the mathematical model of the pinhole camera is explained in detail in Pretehesis.

The `calibrateCamera` function minimizes the sum of squared differences between observed image points and predicted image points, given the current parameter estimates in the camera matrix M. The predicted points are calculated from the corresponding points of the 3D object using the transformation defined by the equation 7.8. This process represents a nonlinear optimization problem, which is solved using the Levenberg-Marquardt optimization algorithm [43].

After a successful calibration, a "CameraMatrix" was obtained, which consists of intrinsic camera parameters and distortion coefficients. These values are fundamental in many subsequent computer vision tasks because they allow the transformation of 3D real-world points into the 2D camera image plane, including our case. They also help reduce or eliminate distortions such as radial and tangential distortions present in the camera lens, which was not necessary in this case because the Intel RealSense D435 camera is already calibrated by the manufacturer [44]. The entire calibration process code can be found in the AppendixB.

The following subsections will discuss the initial configuration of the UR5e robot and illustrate the use of a calibrated end-effector alignment camera.

7.3.2 Checkerboard Detection and End-Effector Alignment

The end-effector of the UR5e arm is equipped with a camera for visual inspection and feedback, as described earlier in Chapter 4. In the final robot pose, this camera obtains a clear view of the cooler and a checkerboard pattern, which can be seen in Figure 7.8. This alignment is critical to ensure precise chamfering of the pipes protruding from the cooler plate.

The visual information, combined with the image processing pipeline used by OpenCV, allows the system to detect the checkerboard and calculate the necessary transformations for accurate and precise pipe recognition for the chamfering process.

After the camera captures an image of the cooler and the checkerboard, the system uses the OpenCV library to detect the corners of the checkerboard in the image. This results in a mapping between real-world 3D points (objpoints) and 2D image plane points (imgpoints). This mapping allows the system to determine the position of the checkerboard in relation to the camera.

The position is calculated using OpenCV's `solvePnP` function [41], which takes as input the `objpoints`, `imgpoints`, the camera's intrinsic matrix and distortion coefficients obtained from the previous calibration, see subsection 7.3.1 . The `solvePnP`

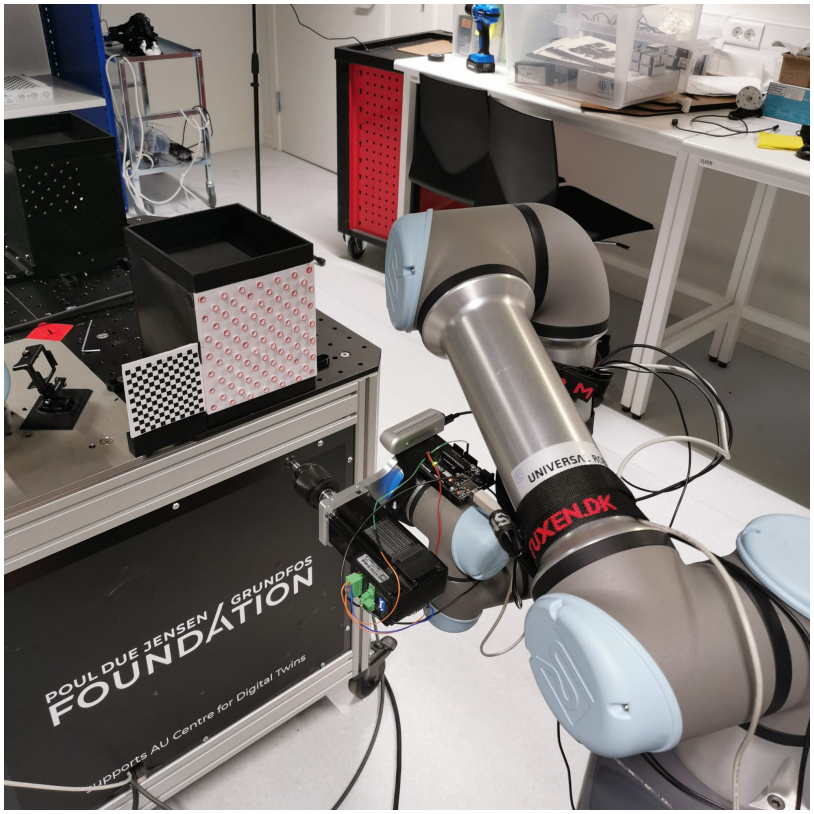


Figure 7.8: The camera's position relative to cooler and checkerboard.

function implements several methods for solving the Perspective-n-Point problem, which is the problem of estimating the position of a calibrated camera given a set of "n" 3D points in the world and their corresponding 2D projections on an image. [45]. The mathematical model behind `solvePnP` is based on the camera projection matrix explained in the Equations 7.6 - 7.8:

The function `solvePnP` gives a rotation vector and a translation vector that together form a transformation matrix that represents the position of the checkerboard in relation to the camera.

The rotation vector is converted into a rotation matrix using the Rodrigues formula [46; 47]. The Rodrigues formula is used to transform a rotation vector (also known as an axis angle representation) into a rotation matrix and vice versa:

$$R = I + sin(\theta) \cdot r^{\times} + (1 - cos(\theta)) \cdot r^{\times 2} \tag{7.9}$$

where I is the identity matrix, $r^{\times}$ is the skew-symmetric matrix form of the rotation vector, θ is the magnitude of the rotation vector, and $r^{\times 2}$ is the square of the skew-symmetric matrix.

The final transformation matrix from the camera to the checkerboard can be represented as:

$$\left[\boldsymbol{camera\_to\_checkerboard}\right] = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{7.10}$$

where $\left[R\right]$ is a 3x3 rotation matrix obtained using Rodrigues' formula and $\left\{t\right\}$ is a 3x1 translation vector obtained from `solvePnP`

It's important to note that the checkerboard plane is assumed to be parallel to the cooler plate, so aligning with the checkerboard effectively aligns the robot with the cooler as well.

After obtaining the transformation matrix from the camera to the checkerboard, the system computes the transformation from the robot base to the checkerboard. The transformation matrix from the end effector to the camera is known from MoveIT HandEye Calibration which is described in Section 7.2. Additionally, the current transformation from the base of the robot to the end-effector is obtained using the `tf` package in ROS [48]. The overall transformation from the robot base to the checkerboard is computed by multiplying these transformation matrices:

$$[\boldsymbol{base\_to\_checkerboard}] = [\boldsymbol{base\_to\_end\_effector}] \times [\boldsymbol{end\_effector\_to\_camera}] \times [\boldsymbol{camera\_to\_checkerboard}] \tag{7.11}$$

Here, the $\times$ symbol denotes matrix multiplication.

At this stage, the rotation part of this transformation matrix is converted back to a quaternion, since `MoveIt Commander` accepts quaternions as orientation input. The system then sets this calculated quaternion as the target orientation for the end effector and moves the robot to align its end effector with the cooler plate. In order to further ensure the most ideal optimal positioning, the last joint of the robotic arm is set at 0 degrees, aligning the end-effector (and the attached camera) in the same plane as the checkerboard and the cooler, and with the floor. This configuration reduces the relative rotation between the camera and the cooler, providing an optimal view for identifying the tubes for the chamfering process. The complete code for alignment of end-effector with the checkerboard can be found in Appendix H

The effectiveness of this matching process depends on the accuracy of the checkerboard detection, the precision of the robot's sensors and actuators, and the correct calculation of the transformation matrices. By visually inspecting the cooling plate, the robot can self-correct and ensure its alignment for optimal level performance. Figure 7.9 illustrates the reference frame used in checkerboard detection.

To clarify the process, a flow chart can be created detailing the steps of capturing an image, detecting the checkerboard, calculating the transformation matrices, aligning the robot's end effector by adjusting its orientation and setting the last joint to 0 degrees, and finally positioning the robot for optimal chamfering.

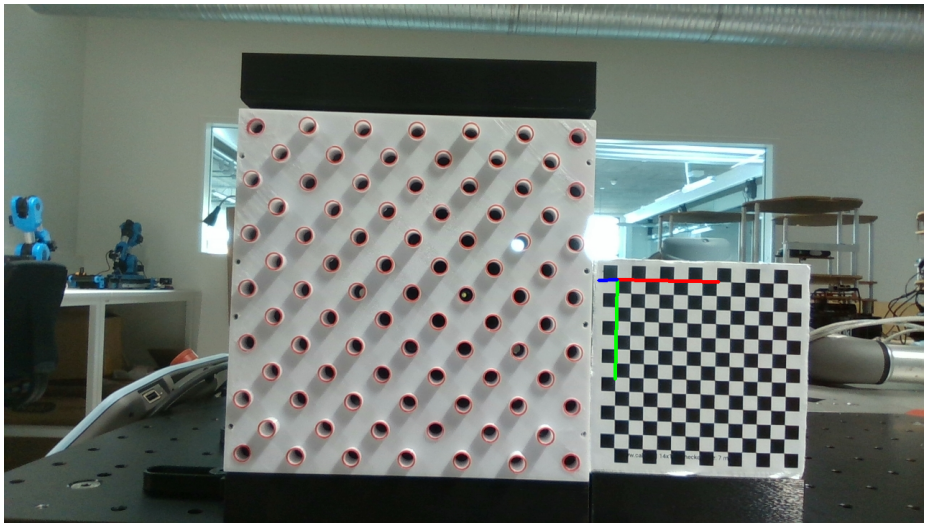


Figure 7.9: The checkerboard frame of reference in the camera's view.

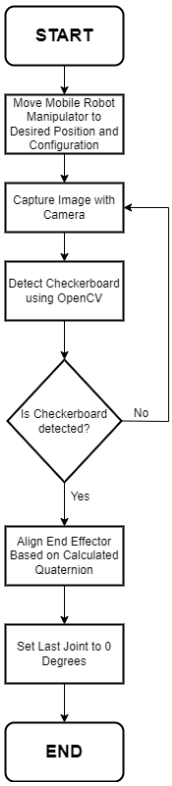


Figure 7.10: Flow chart of the robot alignment process.

7.3.3 Validation and Error Mitigation

Ensuring the validity and accuracy of a robotic system includes testing and appropriate error-handling strategies. Several measurements were performed to verify the reliability and performance of the robotic system.

Calibration Accuracy

The accuracy of the camera calibration process was assessed by performing several calibration procedures and evaluating the camera matrix and distortion coefficients obtained in each run. To account for different imaging conditions, calibrations were performed on several sets of images taken in different scenarios, including low light, high light, isolated environment, and non-isolated environment. Each set of images represented a mixture of these different conditions to ensure the robustness of the calibration process. Finally, the calibration was also performed on a set containing all the images together to see if it is strongly affected if the set contains many more images.

These different calibration procedures allowed a comprehensive evaluation of the system's performance under different recording conditions. Variations in the calibration results were examined, and for more reliable calibration results, the average of the values from several times was taken. This approach mitigates any possible one-time errors or anomalies that may occur during a single calibration.

Variations between camera matrices obtained in different calibration cycles were evaluated by calculating the Frobenius norm of their difference in relation to the average camera matrix [49]. The Frobenius norm, a measure of matrix norm similar to the Euclidean distance for vectors, yields a single scalar value that represents the magnitude of the difference between two matrices. A smaller Frobenius norm indicates a smaller difference, suggesting a more consistent calibration result. The differences for each calibration are visualized in Figure 7.11.

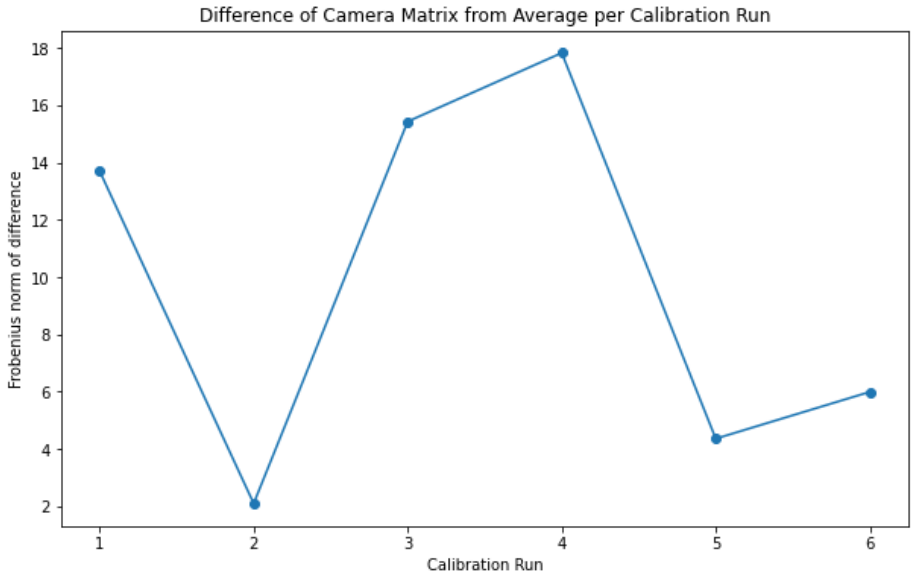


Figure 7.11: Difference of Camera Matrix from Average per Calibration Run

The difference of each distortion coefficient from its average value across the calibration runs was also calculated. These differences for each calibration run and each coefficient is visualized in Figure 7.12.

Calibration accuracy was further evaluated by calculating the Reprojection Error.

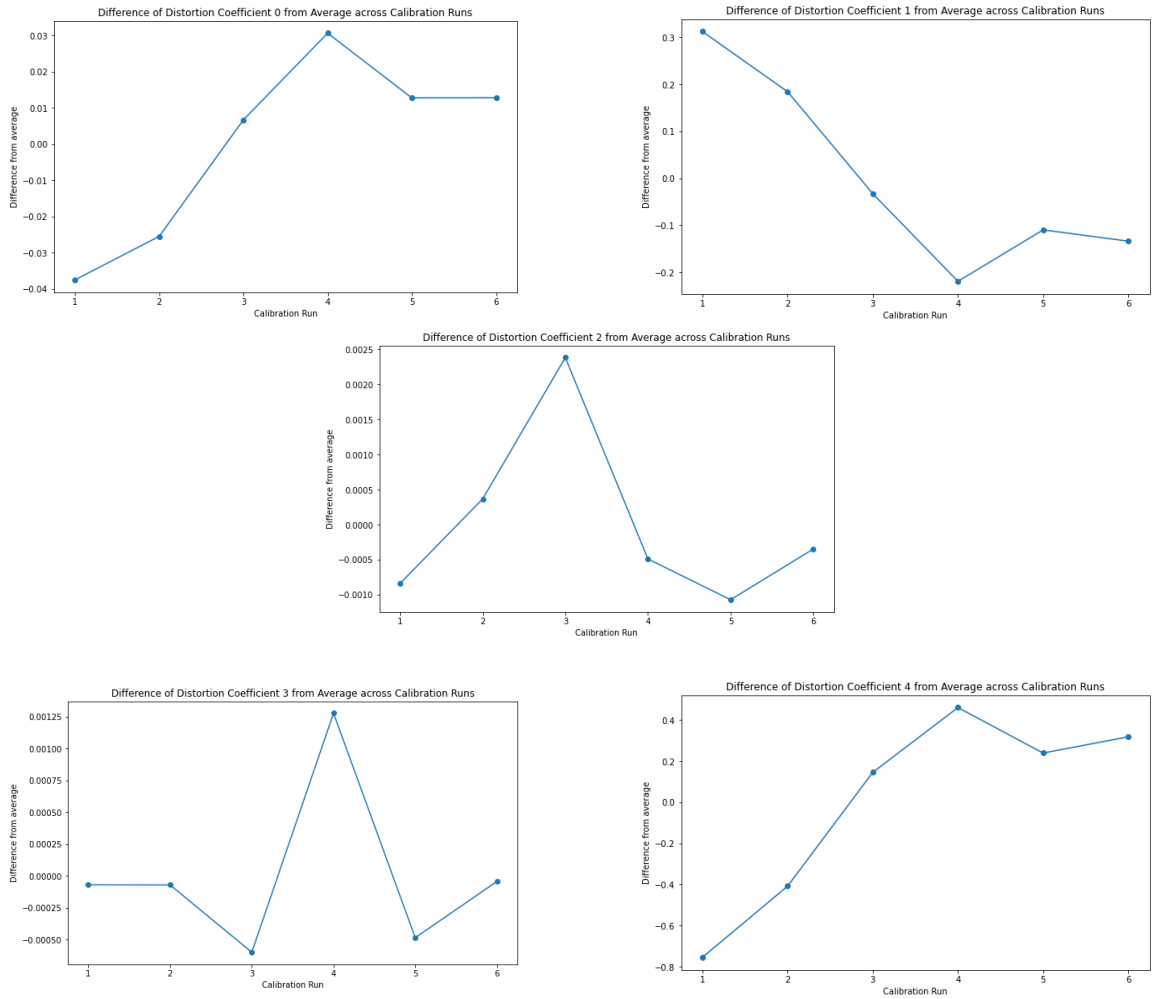


Figure 7.12: Difference of Distortion Coefficients from Average across Calibration Runs.

Reprojection Error is a common metric used to measure the accuracy of camera calibration. It quantifies the amount of error between the observed feature point positions in the image and the projected positions of the same points using the estimated camera parameters [42]. Mathematically, it can be represented as follows:

$$E = \sum_{i=1}^{N} ||x_i - \hat{x}_i||^2 \tag{7.12}$$

where x_i are the observed 2D points in the image, $\hat{x}_i$ are the projected 2D points obtained by projecting the corresponding 3D points X_i using the estimated camera parameters, and N is the total number of points. The projection of the 3D points onto the image plane is computed as:

$$\hat{x}_i = K[R \ t]X_i \tag{7.13}$$

where K is the camera intrinsic matrix, R is the rotation matrix, and t is the transla-

tion vector [50]. This process was achieved by using OpenCV's `projectPoints` and `norm` functions.

A lower reprojection error indicates a more accurate calibration. In this study, the calibration process was carried out six times. After each calibration run, the reprojection error was calculated for all images used in the calibration process.

The mean reprojection error across all six calibration runs was then computed. The calculated mean reprojection errors for each calibration run are plotted in Figure 7.13.

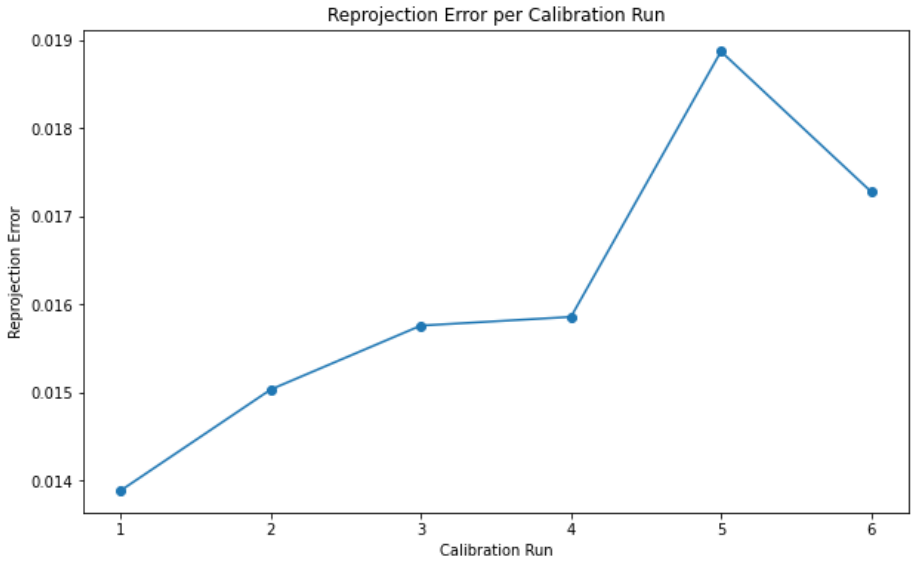


Figure 7.13: Plot of Mean Reprojection Errors for each Calibration Run

As can be seen from Figure 7.13, all reprojection errors were relatively small for all calibration runs, suggesting a successful calibration process. The consistency in the small magnitude of the reprojection errors across all runs further instills confidence in the calibration's accuracy, confirming the robustness of the calibration process across varied imaging conditions.

In this code, `cv.projectPoints` re-projects the 3D points (object points) onto the image plane using the estimated camera parameters. The norm between the observed 2D points (image points) and the re-projected points is computed, normalized by the number of points, and summed up to obtain the total reprojection error.

Alignment Verification

The second part of the validation process involved verifying the accuracy of the end-effector's alignment with the checkerboard frame. This procedure not only focused on the end-effector's rotation but also its position, with the aim of moving the end effector to the center of the checkerboard frame.

Tests were conducted with the checkerboard rotated at various angles. The robot was tasked to align its end-effector with the rotated checkerboard frame. A visual

inspection was carried out and images were taken, which can be seen in Figure 7.14, to confirm whether the end-effector correctly aligned with the rotated checkerboard and reached the center of the frame.

Figure 7.14: Image of checkerboard with associated frame and alignment of end-effector with it

These validation measures and iterative error mitigation techniques ensure the robustness and reliability of the robotic system. It is important to note that no system can be completely error-free. However, constant evaluation and fine-tuning can significantly improve the performance and reliability of the system, paving the way for successful operation, which was attempted to be achieved in this case as well.

The validation and error mitigation process is iterative and forms a key part of the overall system design, ultimately improving the system's ability to perform accurate and consistent chamfering tasks.

7.4 Pipe recognition

7.4.1 Methods

In order to define the most accurate and stable method for finding the position of each pipe from the cooler, multiple methods were tested. First of all, adequate methods had to be chosen based on specifications and the nature of the problem. Since pipes are of simple circular shape, that property could be used to apply existing algorithms, including Hough Circles [51] and Blob Detector [52]. Moreover, more complex methods including Machine Learning algorithms could be used based on deep learning, mainly convolutional neural networks to detect trained objects [53].

Blob Detector

The first promising method tested in order to detect ends of pipes was the Blob Detector function [52]. The main principle of a blob detector is to detect specific shapes based on color differences within the image, where a set of pixels define a specific colony or object which based on color can be distinguished from the surrounding within the image. The algorithm can be used as a method to define inertial shapes such as circular ones, which appeared in image due to the characteristics of pipes. The main parameters that play a role in setting up outputs are area, circularity, inertia, convexity, and threshold, which mainly define the shape of the object to be

found as well as its color scale. Blob detector after adjustment for finding circular shapes is capable of defining ends of pipes in a precise manner, however, it struggles with shadows and is very sensitive to light changes, where it can make mistakes while focusing on darker spots created by shadows of pipes than pipes itself. During tests on chosen images and camera recordings, the algorithm struggled to find all pipes and in most cases, it was defining their eds with offset as it is shown in Figure 7.15. Moreover, the algorithm is very slow in its nature, where each pixel on every frame has to be analyzed with applied filters. Adding the issue of not recognizing all of the pipe ends creates a performance problem while using camera recording. Additionally, a blob detector has also a problem with distinguishing the difference between pipes and other circular shapes in the surrounding, therefore it can be very easily misled, especially if the intruder shape has similar parameters as pipe ends.

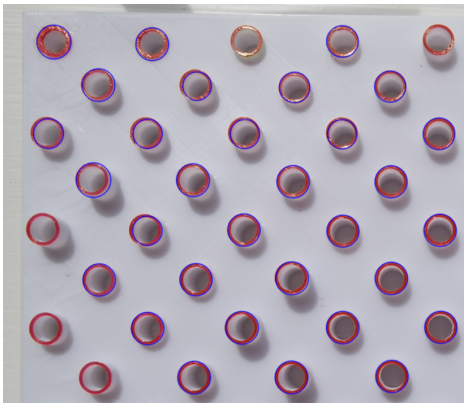


Figure 7.15: Pipes detected by Blob Algorithm

Hough Circles

Hough Circles [51] is an algorithm specifically designed to detect circles on the image. It applies a voting procedure in the parameter space to identify potential circles, which means that it has a more specialized approach to finding circles rather than blobs. The algorithm defines circles based on edge detection and its gradients. This makes this algorithm much faster than Blob Detector. However, same as Blob Detector it has also drawbacks with distinguishing pipe ends from similar surrounding shapes. Moreover, the algorithm has also problems with accurately defining all pipes and very often, even after filtering with thresh-holding and grayscale of the image it is susceptible to detecting too many circles or skipping those important ones in the image or camera frames as it is shown on Figure 7.16.

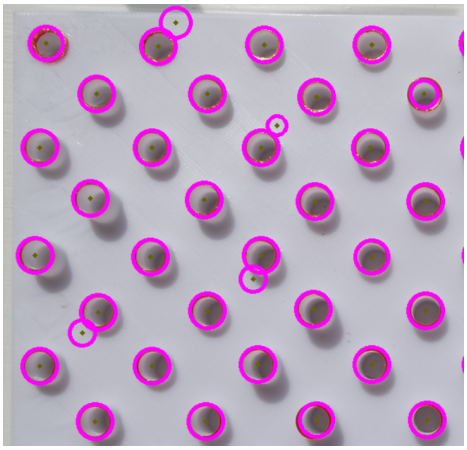


Figure 7.16: Pipes detected by Hough Circles

YOLO

One of the more complex algorithms is YOLO (You Only Look Once) [53], which is part of deep learning, specifically based on a convolutional neural network (CNN). Based on trained models YOLO and defined classes algorithm can detect objects with adjustment of weights and parameters of the neural network such as a number of grid cells and confidence threshold. The main advantage of using an algorithm over previously described ones is that it can distinguish trained objects from surroundings and more precisely and faster detect ends of pipes from multiple perspectives due to confidence level. The algorithm is also very robust due to the fact of finding within a short time all of the objects within a frame, see Figure 7.17.

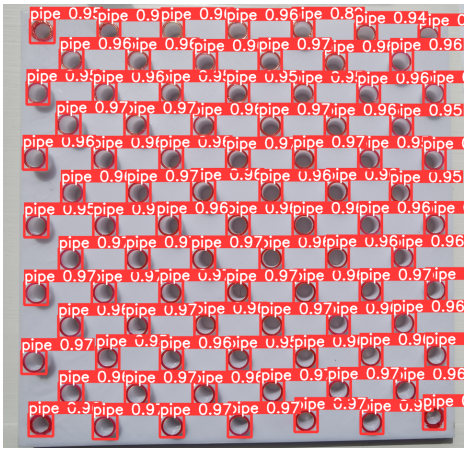


Figure 7.17: Detected pipe ends by YOLO v5 algorithm

7.4.2 Model training and validation

Due to the best results achieved with the YOLO algorithm, it was used as a primary method in defining the positions of the ends of tubes and their image coordinates. To incorporate YOLO to conduct this task, a training model must be created for machine learning. The initial part of is a manual creation of labeled pictures of the

object of interest, therefore multiple pictures of the cooler with pipes had to be taken and labeled manually (Figure 7.18). Several pictures and label objects on them can affect the performance of the algorithm and training [54], generally the higher number of pictures and labels the more robust is model. It is also important to set labels as accurately as possible, meaning that the edges of boxes have to be as close as possible to the edges of pipes in order to define accurately the centers of pipes and reduce noise from surroundings while enhancing the confidence level of the trained model.

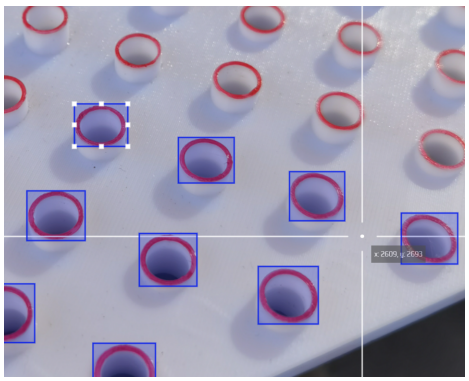


Figure 7.18: Creation of labeled pictures

After the creation of a labeled dataset of pictures, those are used as part of training and validation sets in order to conduct machine learning (around 70% of the dataset is used for training and the remaining for validation [55]). The training set is used to train parameters of models and their performance, the validation set on the other hand is used to assess the generalization and fine-tune hyper-parameters of the model.

Evaluation of the model

YOLO algorithm is a regression-based object detection model, which can be evaluated through multiple metrics to define its precision and correctness [56]. The primary determinant for the performance of the model is the Intersection over Union (IoU) parameter, which is based on the accuracy of bounding boxes between truth and predicted ones during the training. In simple words, if IoU is equal to 1, then it means that bounding boxes of detected object overlap perfectly to the truth-labeled model. Based on the value of the parameter it is possible to establish it as a threshold in order to establish if the prediction is valid or not. If the model has a threshold value of 0.95, then it means that everything below this threshold would be classified as False Positive (FP), which means wrong detection, and everything above the value would be classified as True Positive (TP) meaning that model achieved correct detection. Moreover, when the model fails to detect an object in the presence of ground truth it is classified as False Negative (FN). Based on mentioned parameters it is possible to define Precision and Recall metrics, which can define precision-recall curves. Precision is the proportion of correctly predicted positive instances out of all instances predicted as positive. The recall is the proportion of correctly predicted positive instances out of all actual positive instances. It focuses on the ability of the model to identify all positive instances. Both metrics can be defined based on Equations 7.14 and 7.15

$$Precision = \frac{TP}{TP + FP} \tag{7.14}$$

$$Recall = \frac{TP}{TP + FN} \tag{7.15}$$

Plotting together precision and recall can give a glance at how the model can accurately identify positive instances while minimizing false positives and false negatives. Generally, balanced precision and recall are most desirable. In the case of the plot in Figure 7.19, created during the training of the model for pipe detection it might be noticed that model sustains high precision, while achieving a high recall value, meaning that model is very stable and precise.

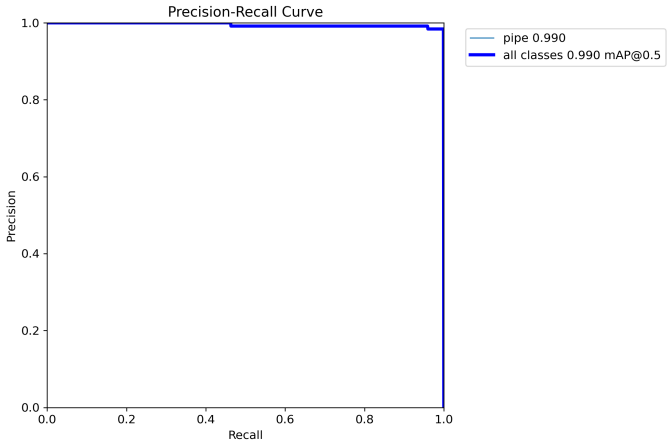


Figure 7.19: Precision-Recall plot

Furthermore, the AP (Average Precision) metric can be established from the area under the precision-recall curve and can be described by Equation 7.16:

$$AP@\alpha = \int_0^1 p(r)\, dr \tag{7.16}$$

Where α is specified as the threshold value. Having all previously described data, it is possible to define the main and common metric, which is Mean Average Precision (mAP), which can be evaluated by equation 7.17:

$$mAP@\alpha = \frac{1}{n} \sum_{i=1}^{n} AP_i \tag{7.17}$$

Metric is described as the average of AP numerical values over all classes. Depending on the value of the threshold the higher value closer to 1, the better model is constructed and can be described as more precise and robust.

While looking at Figure 7.20 it might be noticed that mAP0.5 and mAP0.5-0.95 increase with the time and number of epochs [57] to reach final values respectively

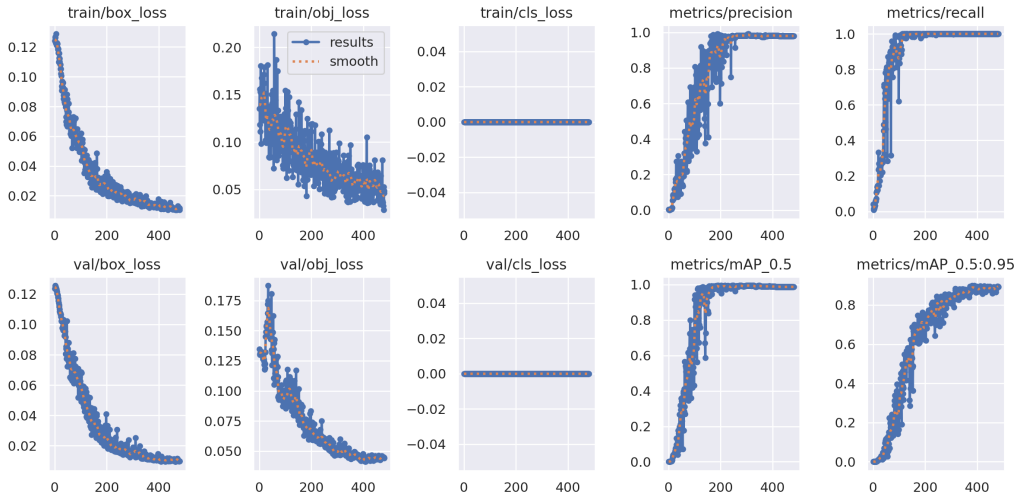


Figure 7.20: General metric parameters from training YOLO model for pipes

0.99 and 0.9. Those values are very close to 1 and they prove that created model is very accurate and precise.

Figure 7.20 represents also plots of train-box loss, train-object loss, train-class loss, and the same for the validation set. Those are loss functions, which penalize the model in the case when it fails to localize the bounding box, not find an object, and not classify the object to the proper class. The last mentioned one is at level zero, since, there is only one class (pipe) in the model. It might be noticed that with a growing number of epoch values tends to go to 0, meaning that with time accuracy and detection abilities of the model are increased and the model is capable of more stable and precise detections.

In the evaluation of the established model, the F1-confidence curve might be used, which describes the relationship between the F1 score and the confidence level of the detecting object. The F1 score is used as a metric to combine together precision and recall in order to establish the best performance of the model. It might be noticed in Figure 7.21 that the peak of the F1 score is appearing at around 0.9 of confidence, meaning that at this value of confidence, the model is achieving the best balance between precision and recall.

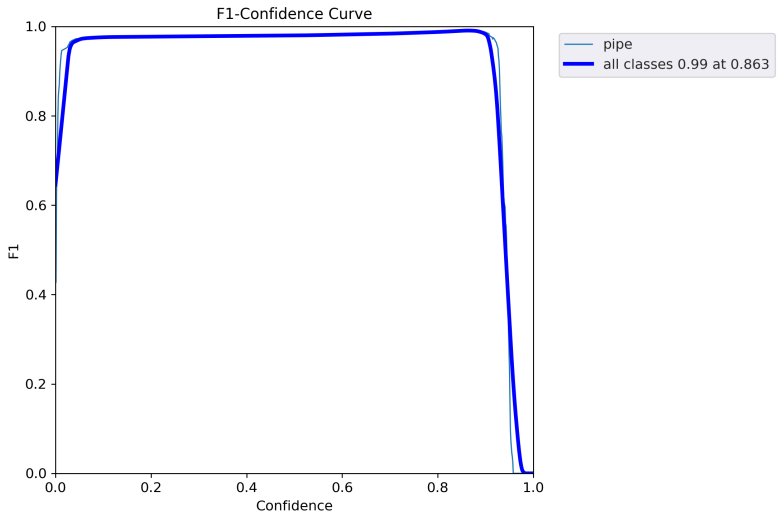
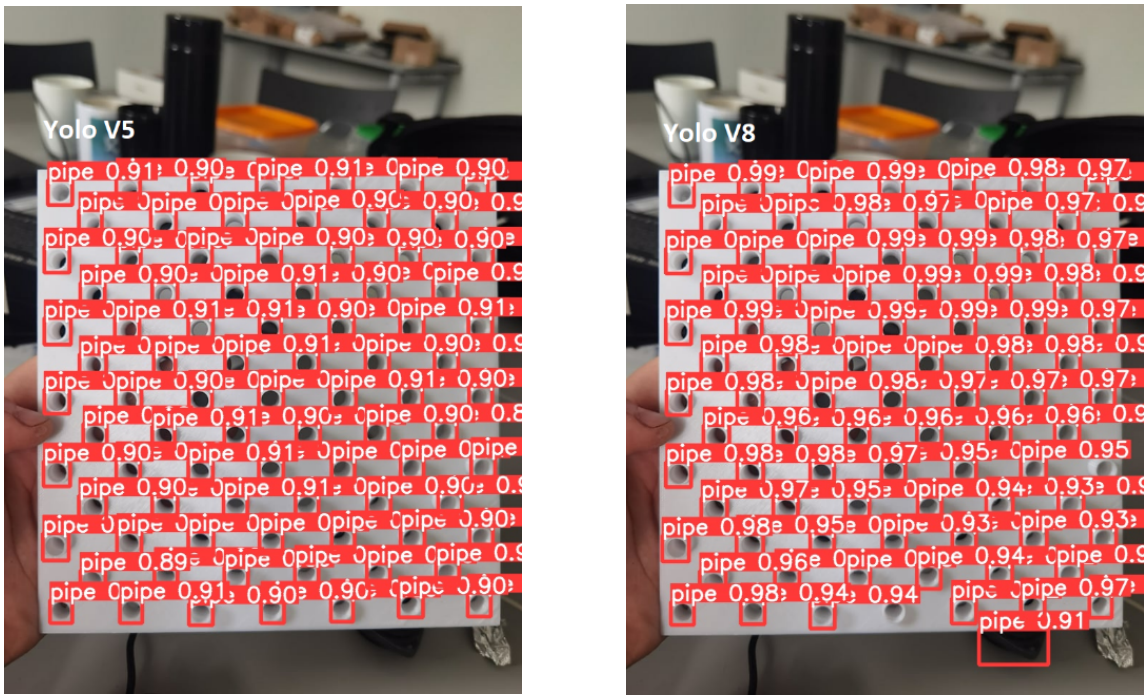


Figure 7.21: F1-confidence curve of the model

Comparison between different YOLO versions

Yolo V5 is known for its simplicity, accuracy, and speed [58]. However, with time new versions of Yolo were developed with improved parameters. The model of detecting the end of pipes with Yolo V5 worked pretty well as it was described in the previous section. To check if results could be enhanced even further, YOLO V8 was trained on the same dataset and compared to YOLO V5 in order to compare both models on different versions. Results might be noticed in Figure 7.22.



a) YOLO V5 b) YOLO V8

Figure 7.22: Comparison of detection betwee

It might be noticed at first glance, that YOLO V5 detected pipes in much better order than YOLO V8. That is an unexpected result since the newer model should be improved and more stable. First of all, YOLO V8 struggled to find all of the pipes within frames, whereas YOLO V5 barely had a problem with it. It can be also noticed that YOLO V8 inaccurately found one of the objects close to the cooler to be a pipe with a huge confidence level.

The same problem appeared furthermore with other testing images, for example in Figure 7.23. At the beginning it was concluded that the model was either under-fitting or over-fitting, however different changes in the dataset could not give reasonable results, therefore further improvement of the model with YOLO V8 was unnecessary, considering that YOLO V5 achieved robust results. It is possible that YOLO V8 could achieve better precision and faster response after a much bigger dataset than was used in the case of YOLO V5. It might be also noticed in Figure 7.22 that YOLO V8 achieved a higher confidence level while detecting some of the pipes, although the model is much less stable.

Figure 7.23: Problems with detection by YOLO V8

7.4.3 Program structure

An established model for determining pipes from frames of camera recording is one of the main steps in camera vision code, however, to incorporate it in the process it is important to create an appropriate program and adjust its functionality so that it could from the initial object detection of pipes provide a list of coordinates, which further could be used to define the position of the tool. The base detecting code of YOLO is indeed detecting objects, however, it needs further processing to adjust it to define the coordinates of pipes since in defaults it does not have those capabilities and therefore additional modules and functions must be added to the main code. What is more YOLO model is detecting pipes randomly, it does not have any sequence or manner in which pipes are detected, basically, the numbering of detected pipes depends on the order in which specific pipe in the cooler was detected. This can be problematic because it can force drill heads to move chaotically to drill ends of pipes. Moreover, detected coordinates are prescribed in position at the image or frame, which is why translation into real perspective is needed [59]. Lastly, all of the results must be saved in a file, to be read by the chamfering program to proceed with the process of chamfering.

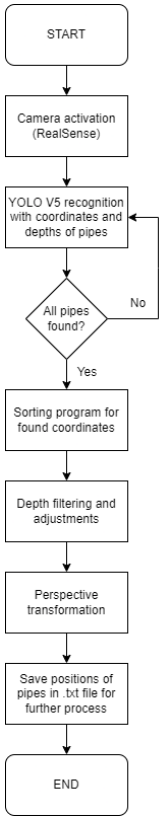


Figure 7.24: Diagram of the program for defining coordinates of tubes with depths for chamfering process

The main structure of the program for achieving coordinates for the chamfering process by camera and YOLO model is represented in Figure 7.24. It contains steps to solve previously mentioned issues and gather needed parameters.

7.4.4 Detection of coordinates and depths

The first modification of the detection code of YOLO includes the incorporation of the RealSense camera (Appendix A.2). The default setting for detection for YOLO is to have a defined path to the image or recording for processing, which afterward is saved in the results. However, by including the RealSense library in the main code, it is possible to set up the YOLO model for the detection of pipes for each frame received from the camera during recording. RealSense library incorporates drivers for both color and depth images, which can be used at any time by the main code to receive data from recording. Receiving and processing each frame depends on the number of frames per second for processing (default 30 FPS), which might cause problems with the smoothness of recording. However, because the end-effector is fixed at the same position and is not moving, it requires at least one frame, very likely the first one received from the camera, just to detect all pipes and stop recording for further processing of results.

After activation of recording YOLO model tries to create predictions, meaning that algorithm is trying to define multiple boxes and objects within the analyzed frame as it might be noticed in the "while true loop" in Appendix A.3. After detecting objects YOLO model is trying to adjust the box size for them with the Non-Max Suppression algorithm (NMS) [60]. Predictions are then stored by the YOLO model and saved (lines 43-58 in Appendix A.3).

After successful detection in the frame from the camera, image coordinates X and Y are then defined for each of the detected boxes. At the same time depth for each pipe and the depth to the plate next to the pipe is defined using defined at the beginning position of the pipes. Each coordinate of pipe is defined with the usage of box-utils library [61], which defines two corner points of bounding box for each of detected object. Based on those coordinates it is possible to define the middle of the box (line 14 in Appendix A.4). The defined coordinate can be used to define the depth distance between the tip of the pipe and the plate. Depth to the tip of the pipe is defined based on the ROI mask (Region of interest) (line 20-21 in Appendix A.4), where the average depth within the circular region around the center point of the detected pipe is calculated. ROI mask aims to improve the detection of the distance to the pipe by detecting multiple samples within the area of interest and then based on user specification output value, which is the closest to requirements. It is worth mentioning that it is very hard to define the distance from the camera to the edge of the pipe, due to the very small area that can be detected, therefore most of the time, distance cannot be established due to accuracy problems, which are limited due to abilities of depth sensor of RealSense camera. Even though the ROI mask approach improved results, yet still could not always provide acceptable output. To support depth recognition, the second function is defining it, by looking for depth to the plate, which is set to points offset from pipe centers (line 22 in Appendix A.4).

Depth to the plate of the cooler is much easier to be detected due to the smooth surface and bigger area that can be detected. That is why in the main code, though the distance to the pipe was established, distance to the plate was mainly used due to its robustness and more predictable results.

Detecting code is specified to break a loop for catching frames and looking for objects, while it defines the specified number of pipes that it is supposed to determine (lines 3-4 in Appendix A.6). After detecting objects post filtering is appearing to make sure that all detected values, especially for depth are acceptable for each of the pipe. In some cases, the depth camera can be not capable of detecting all depths for specified coordinates, therefore it is going to provide a not-a-number value. Those values are cleared by exchanging them for zero. However, at this point still, some of the data can produce wrong output and create problems, which is why additional filtering techniques were used.

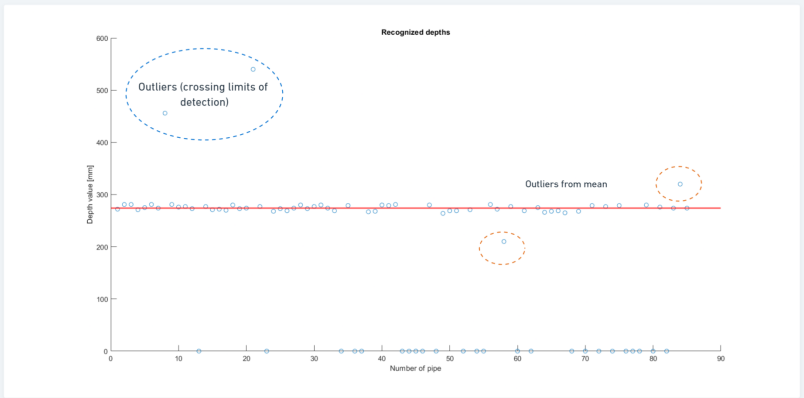


Figure 7.25: Depth detection and filtering

Taking as example data received after one of the tests, which are at the plot in Figure 7.25, it might be noticed that in the dataset there are few outliers, which can cause stability issues and problems with further processing of coordinates. Therefore threshold value of 400 [mm] is used since the camera operates between 15 [cm] and 40 [cm] range to get the best possible values of depths [62]. If any value read from samples crosses the threshold value, meaning that is not in the range it is automatically removed from the dataset without any further consideration in mean calculations. Further processing includes the removal of outliers, which are crossing the threshold value from the calculated mean value (Figure 7.25). All coded filtering operations might be noticed in Appendix A.6. This provides even further filtering and improvement of the mean value of detected depth and gets as close to the real value.

7.4.5 Sorting of coordinates

As it was mentioned in the previous section, the Yolo algorithm does not have any specific order in which it detects objects. If there is for instance image with pipes, the algorithm would not find objects from one side to the other saving one by one. Instead of saving data in a specific manner, the algorithm saves objects by prioritizing

the first spotted objects in the image, therefore if the first detected pipe would appear in the middle of the cooler, then it would be assigned as number one in the list.

In order to accelerate the process of chamfering, so that the end-effector does not have to take the long path between pipes, coordinates are sorted in the manner so that they can be chamfered row by row from the top of the cooler to the bottom as shown in Figure 7.26.

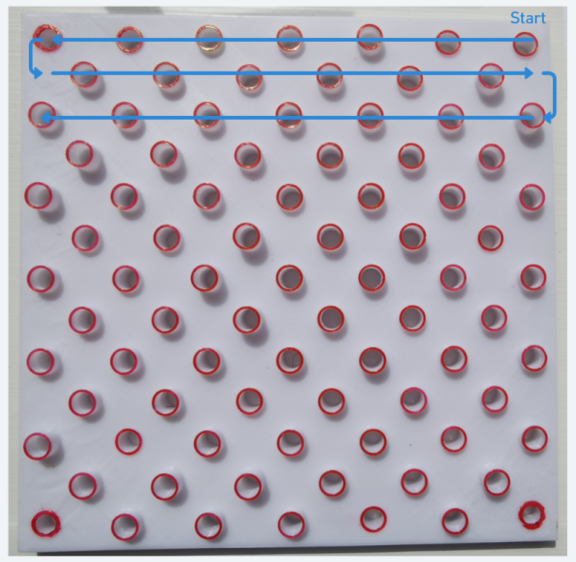


Figure 7.26: Pattern of moving drill head

Pattern, which is presented in Figure 7.26 represents one of the fastest ways to move the drill heads to reduce the time needed for movement, also it is a representation of how pipes are chamfered manually by the worker. Movement could start from any corner of the cooler and could follow motion either by row or column.

One of the ways to sort coordinates in this manner is to sort coordinates into sets based on similar values of Y positions (pipes in the same row). Of course, due to small errors within the accuracy of detected coordinates, the threshold must be set up in order to include in set pipes which are within the same value of Y. Sets are numbered based on Y position from top to bottom of the cooler. Afterward, coordinates must be sorted within each set in order based on the X value of coordinates. Therefore if a set has a prescribed odd number then coordinates are sorted based on decreasing manner of X values. If the set has an even number then coordinates are sorted in the growing manner of X coordinate. By using this specific way of sorting drill head can follow the path in the same aspect as it is shown in Figure 7.26 by the blue curve.

The procedure described above was programmed in Python and the source code can be seen in Appendix C.1. Moreover, after segregating coordinates, they are saved in a text file for further processing.

7.4.6 Perspective transformation

As mentioned at the beginning of the chapter, coordinates obtained with the usage of the YOLO model represent the position of objects at the detected image, therefore they do not represent coordinates in the real world and they need a perspective translation in order to orientate chamfering tool in space. In order to translate both Cartesian X and Y coordinates some of the parameters must be determined.

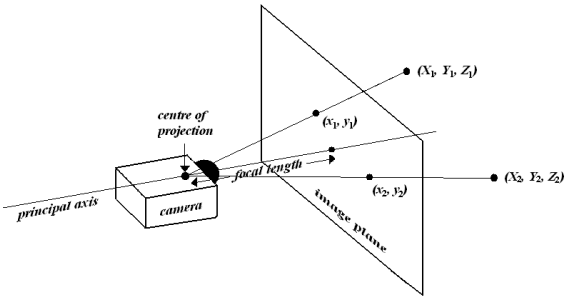


Figure 7.27: Perspective transformation - diagram [63]

First variable is a focal length (Figure 7.27), which is used as a scaling factor for x and y directions in pixels for image [64] and can be described as relationship between both X_{image} and Y_{image} coordinates of image and coordinates of a point in 3D measuring from origin of centre of projection $v = [u, v, w]^T$ with Equations 7.18 and 7.19:

$$X_{image} = f_x \cdot \frac{u}{w} \tag{7.18}$$

$$Y_{image} = f_y \cdot \frac{v}{w} \tag{7.19}$$

Where f_x and f_y are specified as focal lengths for x and y.

Additionally, the principal point of the camera is needed, which specifies the intersection of the optical axis and the image plane.

Both variables can be received from RealSense camera driver with calling function `intr` (line 35-41 in Appendix A.7).

Established variables, as well as depth (Z coordinate) with the usage of the depth image, can be therefore used in order to define the X and Y position of each pipe to be chamfered. To conduct the procedure each image coordinate previously defined by the object detection model must be transformed with Equations 7.20 and 7.21 for respectively X and Y coordinates.

$$X = Z \cdot \frac{X_{image} - C_x}{f_x} \tag{7.20}$$

$$Y = Z \cdot \frac{X_{image} - C_y}{f_y} \tag{7.21}$$

Where C_x and C_y represent the coordinates of the principal point of the camera. To calculate the transformation for each pipe, a function in Python was created, which can be seen in Appendix C.2

7.4.7 Saving of data

The last part of the algorithm for the detection of pipes and receiving their coordinates is related to saving achieved data for further processing. The main diagram representing the whole procedure and dependence of achieved outputs can be seen in Figure 7.28.

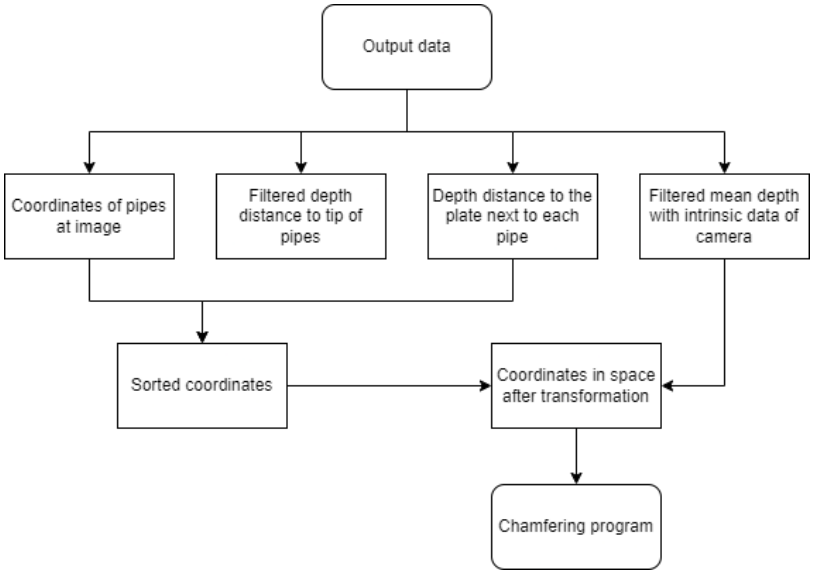


Figure 7.28: Output data from program

The first line of data after output represents data that is received from the model and depth detection. Each of the four sets of data is saved in a text file (code in Appendix A.9).

Saved coordinates of centers of pipes and depth distances to the plate are then used by the Sorting function (Section 7.4.5). Sorted coordinates with coo-responding depths are saved in `pipe-segregated.txt`.

The last step before releasing calculated coordinates is to transform them into real space (Section 7.4.6), which requires sorted coordinates and also the mean value of depth in the case of some of the pipes, the initial depth was not recognized. The final file called `pipe-coord-real.txt` containing transformed coordinates of pipes and their depths is used by the chamfering program.

7.4.8 Analysis of the computer vision system's ability to recognize pipes

The main concern about the computer vision system was to determine how accurate and precise it is including both calibration and recognition of coordinates of pipes

and depths.

Alignment of read coordinates with perfect ones

The accuracy of detected coordinates X and Y of detected pipes were tested based on the mean error value, between detections and the perfect model.

Knowing real measured values and perfect coordinates of the cooler, it was possible to compare their position to those achieved from the pipe detection program.

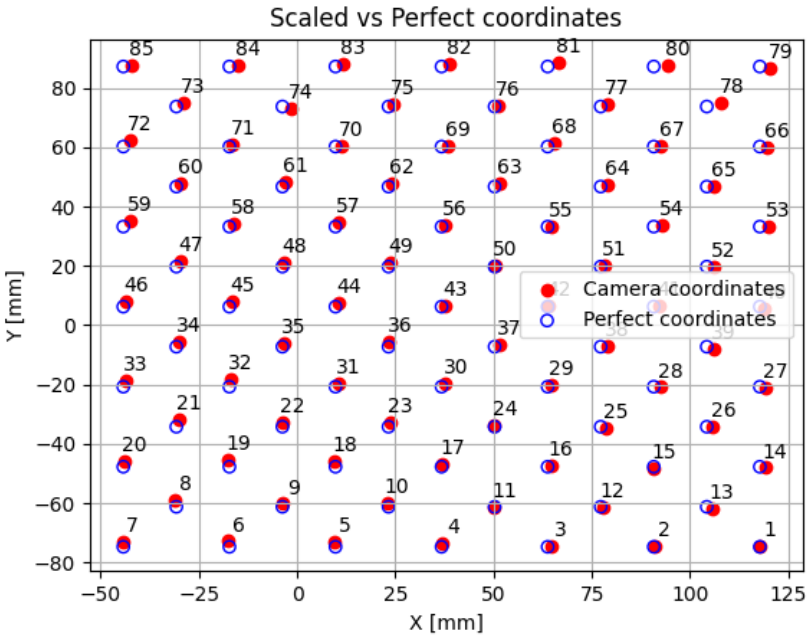


Figure 7.29: Output data from program

Figure 7.29 represents a comparison between perfect coordinate positions (blue) and coordinate positions achieved by pipe detection with camera vision (red). Coordinates are aligned with respect to each based on the alignment of the first corner coordinate.

Furthermore, knowing the coordinates of both perfect and detected coordinates it was possible to establish mean error value based on Equation 7.22:

$$ME = \frac{\sum_{i=1}^{n} \sqrt{(x_{p,i} - x_{d,i})^2 + (y_{p,i} - y_{d,i})^2}}{n} \tag{7.22}$$

Where subscript p and d for coordinates X and Y are described respectively as perfect and detected ones. Number n denotes the number of measurements, in this case, the number of detected pipes.

Based on described above method, multiple tests with the usage of camera vision were conducted in multiple conditions, including light changes as well as different distances to coordinates. The average mean error nonetheless was around 1.61 [mm].

Depth accuracy

One of the parameters that were tested was depth and the accuracy of measurements. Recognized depth, meaning distance between the camera and pipes is based on the depth sensor that the RealSense camera is equipped with. In order to check the accuracy of the sensor multiple tests were conducted including only the RealSense depth sensor as well as tests with an additional laser sensor (Dewalt DWHT77100-XJ). The additional sensor was mainly used to assure that recognized depths are detected properly and there is no big offset between parameters from the pipe detection program and real values. The main testing procedure and checking of average error of measurements were conducted based on the testing setup.

Based on provided specifications about the location of origin of the reference system of camera [65] and carefully measured distance to the place from the camera center it was possible to compare achieved results from the depth sensor to manually measured, see Figure 7.30.

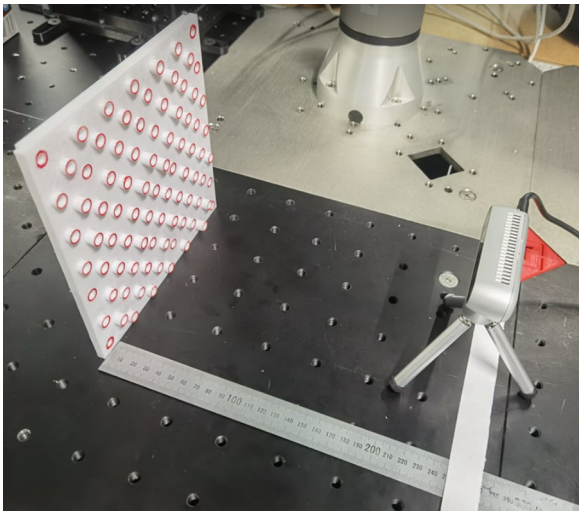


Figure 7.30: Testing station for depth

Since the pipe detection program gathers depths to multiple points in the cooler, therefore the average of detected distances was compared to the measured by ruler ones. Fifteen samples were gathered and compared to measured manually distances between the plate and the camera.

Number	Sensor [mm]	Measured [mm]	Difference		
1	301	300	1		
2	302.5	300	2.5		
3	300	300	0		
4	302	300	2		
5	301	300	1	ME 300	1.3
6	255	250	5		
7	256	250	6		
8	258	250	8		
9	281	250	31		
10	251	250	1	ME 250	10.2
11	401	400	1		
12	404	400	4		
13	405	400	5		
14	406	400	6		
15	403	400	3	ME 400	3.8
		Total ME	5.1		

Table 7.1: Testing station for depth

Based on Table 7.1 it might be noticed that the total mean error of measurements is around 5.1 [mm]. However, based on different distances between the plate and the camera it was discovered that the lowest mean error of 1.3 [mm] is at the measured distance of 300 [mm], meaning that the depth sensor from the camera gives the best outputs being at the previously mentioned distance. It is worth to mention that, when the camera was placed too close to the place, being at around 15 [cm] from the plate, readings very often were not achieved, distorted, or were fairly offset, which means that the most optimal distance for camera to be placed from the plate is between 30 and 40 [cm].

Additional measurements were taken with the laser distance sensor before starting the procedure of chamfering to make sure that the measured distance is correct, see Figure 7.31.

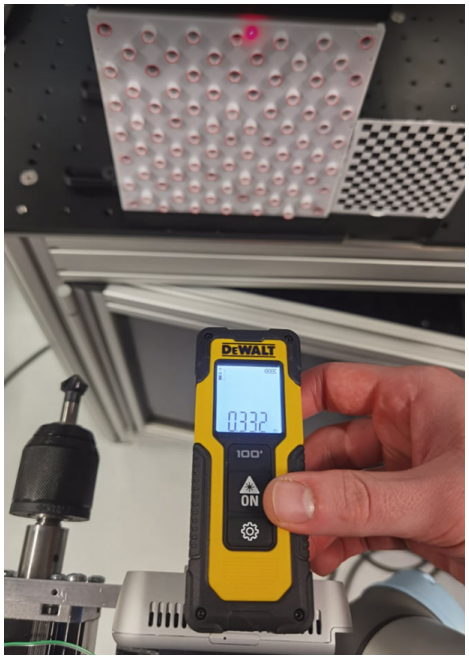


Figure 7.31: Checking distance from depth sensor with laser measure

In most of the cases, measurements were offset by a maximum of 5 [mm]. The error can be produced from multiple sources, mainly caused by the placement of the Dewalt sensor precisely at the origin of the camera. Moreover, errors can appear from the transformation matrices of the robot as well as misalignment of the end-effector.

7.5 Summary

In regards to eye-in-hand calibration, the group concludes that the obtained results are good enough to continue the testing of the final procedure but in case of implementing the procedure for industrial purposes, the camera holder should be designed in a more secure way, which would not allow slightest camera movement while sitting on end-effector, where the group believes possible error occurred in the project. That is not the case for the camera alignment procedure where the error/difference between different tests was not significant.

Described above study about calibration and model for pipe detection for coordinates together with testing provides the conclusion that used methods and provided equipment in the pre-process of chamfering supplies adequate tools to proceed with drilling. Although the technique needs adjustments and an even better-trained machine learning model to reduce the mean error of detected center of coordinates and depth values, it still provides a new general technique and method that can be used in the future for industrial purposes.

8 | Robot Programming and Movement

8.1 UR5e

The robot manipulator UR5e, which is described in Section 3.3, can be controlled in several ways within the ROS (Robot Operating System) environment. One common method is to directly control the robot's joint controllers, which involves publishing the desired joint angles or velocities to the appropriate ROS topics to which the robot's controllers are subscribed. This method provides a high degree of control over the robot's movements but requires a thorough understanding of the robot's kinematics and dynamics. Another method is to use action servers, which allow more complex operations such as path execution. This involves sending the path action goal to the robot's action server, which then controls the robot to execute the path. Furthermore, there are prepackaged Python packages, such as MoveIt Commander, which provide a high-level interface for controlling the robot, and MoveIt Commander was chosen in this case to control the UR5e robot.

8.1.1 MoveIt Commander

MoveIt Commander is part of the MoveIt package that provides a comprehensive set of tools for motion planning, manipulation, 3D perception, kinematics, control, and navigation in ROS [66]. It allows creating a robot model based on a URDF file and defining a `group` that represents a set of joints for planning and control. This group can be easily managed using MoveIt Commander, which provides a simple command line interface to control the robot. The MoveIt package was chosen to control the UR5e robot due to its comprehensive features and ease of use. It is particularly useful in situations where there are complex processes and where the environment of the robot is not ideal, but many aspects should be taken into account when planning the path of the robot, such as awareness of collision with itself and the environment. For this reason, this package was chosen because the automatization of the chamfering process that is being attempted to achieve is really complex and requires a lot of "awareness" of the robot.

8.1.2 Node Initialization

As explained before, in ROS nodes communicate with each other by sending messages, so to control the robot using MoveIt Commander, a new node had to be initialized. This is done using the python function `rospy.init_node()`, where the node name is passed as an argument. This function initializes nodes and registers within the ROS master.

After the node is initialized, MoveIt Commander can be started, and this is done using the function `moveit_commander.roscpp_initialize()`, which initializes the

`ROS C++` interfaces that MoveIt Commander relies on.

Furthermore, a `RobotCommander` object is created, which serves as the interface of the robot as a whole, and it provides information about the robot's kinematic model and the current state of the joints. After that, it is necessary to create the `MoveGroupCommander` object, which is the interface for the planning group created in the MoveIt package, in this case, UR5e, and with its help, the robot manipulator can be controlled by setting the desired poses.

8.1.3 Publishers and Subscribers

The MoveIt Commander node publishes commands in the `/joint_trajectory_action` topic. This topic is subscribed to by the robot control node, which translates these commands into actual robot movements.

On the other hand, the MoveIt Commander node is subscribed to the `/joint_states` topic, which is published by the robot controller node. The `/joint_states` topic includes the robot's current joint positions, speed, and effort. This allows MoveIt Commander to monitor the state of the robot and adjust its commands accordingly.

8.1.4 Control of the UR5e manipulator

The UR5e manipulator is controlled by a set of commands provided by the MoveIt package. These commands make it possible to control the robot based on the state of the joints or based on the Cartesian system. [67]

The main commands used in this project are:

- The command `group.set_pose_target()` - used to set the desired position of the end-effector relative to the robot base. This command takes into account the kinematic structure of the robot and automatically calculates the necessary joint configurations to achieve the desired pose.
- The command `group.compute_cartesian_path()` - used to compute the Cartesian path for the end-effector. This is particularly useful when the robot needs to follow a specific path in the workspace, such as linear motion.
- The command `group.set_joint_value_target()` - used to set a specific robot configuration. It is very useful in the case of the need to set the optimal configuration for some kind of process, which is most conducive to the given task.

To control the end-effector relative to the base frame or any other frame, it is necessary to calculate the transformation between the frame and the end-effector frame. This involves a series of transformations based on the robot's kinematic structure and current joint values.

8.2 MiR200

The provided code represents a simple yet efficient implementation for controlling a MiR200 robot using the Robot Operating System (ROS). It communicates with the robot through a series of published and subscribed topics, adhering to the ROS philosophy of loose coupling and distributed processing.

The main program can be divided into several parts:

8.2.1 Node Initialization

At first, the ROS node is initialized in the constructor of the `MiR200Controller` class:

```
rospy.init_node('mir200_controller')
```

This creates a unique node in the ROS network that will communicate with the MiR200 robot.

8.2.2 Publishers and Subscribers

The controller initializes two main communication interfaces with the MiR200 robot: a publisher and a subscriber. The publisher sends velocity commands to the robot via the `/cmd_vel` topic. The subscriber receives odometry information from the robot via the `/odom` topic. This information includes the robot's current position and orientation.

8.2.3 The Odometry Callback Function

Whenever an odometry message is received, the `odom_callback` function is invoked, which stores the robot's current pose (position and orientation).

8.2.4 The Move Distance and Rotate Methods

These methods use the odometry information and publish velocity commands to control the robot's movement. The `move_distance` method makes the robot move a specified distance at a specified speed, while the `rotate` method makes the robot rotate a specified angle at a specified speed.

The controller continuously publishes velocity commands until the robot has moved the required distance or rotated the required angle. The distances and angles are calculated using the `calculate_distance` and `calculate_angle_difference` methods, respectively.

8.2.5 Distance and Angle Calculations

These calculations are based on simple yet fundamental mathematical equations. The Euclidean distance, d, between two points in a plane, $P1(x_1, y_1)$ and $P2(x_2, y_2)$, is calculated as:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{8.1}$$

The angle difference is calculated using a modification of the principle of modulo arithmetic, which finds the shortest difference between two angles, considering that angles wrap around at 2π radians:

$$\Delta\theta = |(\theta_1 - \theta_2 + \pi) \mod 2\pi - \pi| \tag{8.2}$$

8.2.6 Testing the Controller

In the main function of the program, an instance of the `MiR200Controller` class is created, and then used to move the robot forward by 1 meter, move it backward by 1 meter, and finally rotate it by 90 degrees.

8.3 Mobile Robot Manipulator

The mobile robot manipulator, provided by the university, is a complex system composed of the UR5e manipulator, the MiR200 mobile robot and the ER box, which are described in Section 3.3. This system is equipped with additional components, including a drill and a camera, which are attached to the UR5e end-effector. Given that the goal is to automate the chamfering process using this system, in order to achieve this, the entire system is managed within the ROS environment. As mentioned in the sections above, the UR5e is controlled using the MoveIt package, while the MiR200 is controlled using special commands inside ROS. This section will deal with the creation of the MoveIt package of the mobile robotic manipulator based on URDF explained in Section 3.5, establishing the communication between systems and control of the mobile robot manipulator to automate the chamfering process.

8.3.1 MoveIt Package Creation

The MoveIt package for the Mobile Robot Manipulator is created using the MoveIt Setup Assistant. MoveIt Setup Assistant is a graphical user interface (GUI) tool included in the MoveIt package. It is a very powerful and useful tool that guides the user through the process of setting up a new MoveIt configuration package for the robot. It provides an easy way to configure all the essential elements required for a working MoveIt setup [66].

The first step in the MoveIt Setup Assistant is to load the URDF file. This allows the assistant to generate a semantic representation of the robot, which includes the definition of robot `group` for planning, by defining the joints that are part of the planning group. In this case, a group is created for the UR5e manipulator, which allows it to be managed as a single entity within the MoveIt environment. Figure 8.1 shows the interface with the loaded URDF model of the mobile robot manipulator.

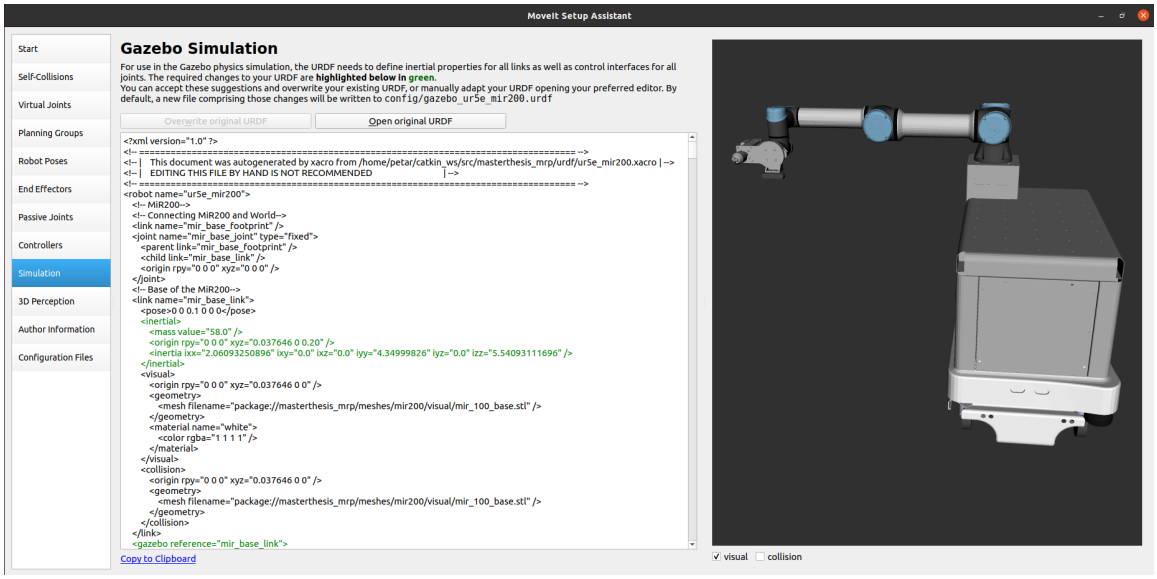


Figure 8.1: MoveIt Setup Assistant Interface

Some of the important steps are connecting to the world frame that serves as a reference frame for all other frames in the system. This is important to ensure that all movements and transformations are calculated correctly. Furthermore, it is possible to define end effectors, which then enables controlling even more special end effectors if there is a need. Also, the passive joints in the robot are defined, which in this case are MiR200 caster wheels. Passive joints do not move but can move due to external forces. Including these joints in the URDF allows the MoveIt package to accurately simulate the motion of the robot.

MoveIt Setup Assistant allows the user to define the controllers needed to control the robot's joints, as for the UR5e, but in this case, the controllers for the UR5e and MiR200 are used through already prepared drivers that are explained in the section. But since the plan was to perform a virtual simulation, that is, to have a digital twin robot in the Gazebo, the option of creating parts in the URDF necessary for running the model in the Gazebo was also used, but this will be explained more in Chapter 9.

After all the necessary configurations were made, the MoveIt Setup Assistant generated the MoveIt package. This package includes all the configuration files needed to control the mobile robotic manipulator within the MoveIt environment.

8.3.2 Launch File Creation and System Communication

The launch file is a crucial part of the ROS system because it is responsible for starting the nodes and setting up communication between them, which is an inevitable part of any robotic system to function within ROS. The launch file for the Mobile Robot Manipulator was created to run the UR5e and MiR200 drivers, the MoveIt package, and the RViz visualization tool.

The UR5e robot driver is launched using the file `ur5e_bringup.launch` from the package `ur_robot_driver`, which is provided by Universal Robots, but modified with the change of the description file (URDF) related to this case. The IP address of the robot is passed as an argument to establish a connection. In addition, a kinematics configuration file is loaded to provide accurate kinematic solutions for the UR5e, which is obtained through integrated calibration.

Similarly, the MiR200 robot driver is launched using the `mir.launch` file from the `mir_driver` package, which is also provided by the MiR manufacturer, but as with the UR5e, the description file has been changed to use the URDF of the mobile robot manipulator. The hostname of the MiR200 is passed as an argument to establish the connection.

The MoveIt package for the Mobile Robot Manipulator is launched using the file `move_group.launch`. This file starts the MoveIt nodes and loads the configuration files generated by the MoveIt Setup Assistant.

Finally, the RViz visualization tool is launched using the `moveit_rviz.launch` file. This file launches RViz with pre-configured settings for visualizing the mobile robot manipulator and its movements.

Communication between nodes is established through ROS topics. The UR5e and MiR200 controllers, which are part of drivers, publish the state of the robot in their topics, which are subscribed to by MoveIt nodes. MoveIt nodes in turn publish planned movements to the `/joint_trajectory_action` topic, to which robot drivers are subscribed. This allows MoveIt nodes to control the robot's movements.

The launch file is shown in Figure 8.2:

```
<?xml version="1.0"?>
<launch>
  <!-- Load UR5e robot drivers -->
  <include file="$(find ur_robot_driver)/launch/ur5e_bringup.launch">
    <arg name="robot_ip" value="192.168.12.100" />
    <arg name="kinematics_config" value="$(find masterthesis_mrp)/config/my_robot_calibration.yaml" />
  </include>
  <!-- Load MiR200 robot drivers -->
  <include file="$(find mir_driver)/launch/mir.launch">
    <arg name="mir_hostname" value="192.168.12.20" />
  </include>
  <!-- Load MoveIt -->
  <include file="$(find new)/launch/move_group.launch"/>
  <!-- Load RViz -->
  <arg name="use_rviz" default="true"/>
  <include file="$(find new)/launch/moveit_rviz.launch" if="$(arg use_rviz)">
    <arg name="rviz_config" value="$(find new)/launch/moveit.rviz"/>
  </include>
</launch>
```

Figure 8.2: Launch file for the system

To visualize the communication between nodes and topics, the `rqt_graph` tool in ROS can be used. This tool provides a graphical representation of the nodes and topics in the ROS system, along with the connections between them. Figure 8.3 shows the `rqt_graph` output for the mobile robot manipulator system.

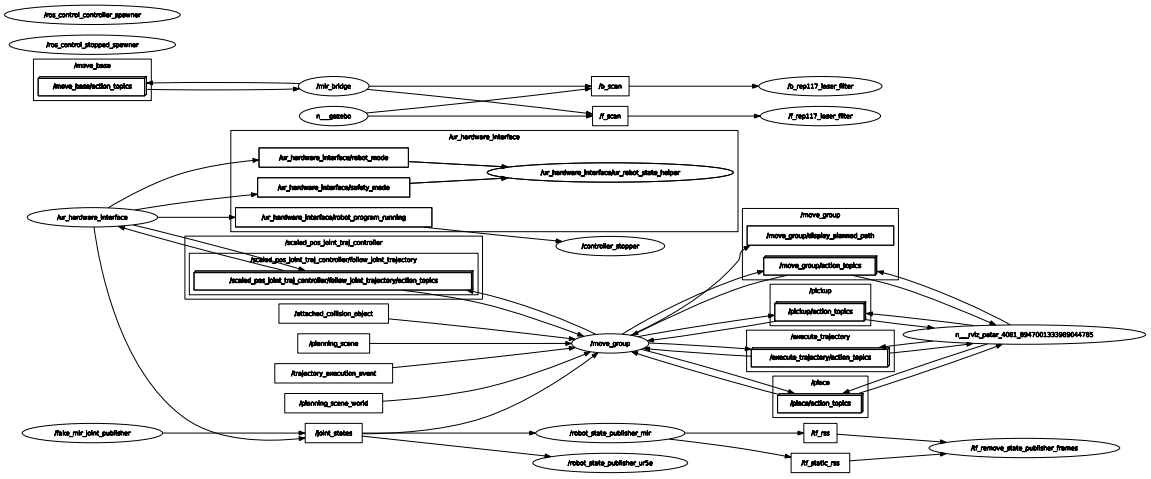


Figure 8.3: Communication between Nodes and Topics

As shown in the Figure 8.3, the UR5e and MiR200 drivers, along with the MoveIt package, are represented as nodes. The arrows between nodes represent the topics they publish and subscribe to. This graph provides a clear overview of how the different components of the mobile robot manipulator system communicate with each other within the ROS environment.

8.3.3 Control of the Mobile Robot Manipulator for Automation of Chamfering Process

As explained through Thesis, automating the chamfering process is a complex task that involves the integration of two robotic systems, the MiR200 mobile robot and the UR5e manipulator, along with additional components such as a 3D camera and a drill. The MiR200 is tasked with moving and positioning the UR5e, while the UR5e performs the chamfering operation. Control of these two systems is achieved through a combination of programming and real-time adjustments, which are made based on feedback from sensors and components.

Although the MiR200 and UR5e are part of the same system, they are managed separately. Each robot has its own controller, which works within individual drivers. Given the nature of the process, it is not necessary for the MiR200 and UR5e to operate in a synchronized or coordinated manner. Instead, each robot must complete its task before the other can continue. As explained in the previous sections, each controller can be programmed using Python. The main program, `main.py` (Appendix K), also written in Python, is based on the process flow described in Section 3.2. This program executes all the modules needed to automate the chamfering process.

In order to ensure the correct functioning of all the necessary modules, it is important to have precisely defined frames for each joint and component involved in movement and calculation. This is key to achieving accurate positioning within the workspace and in relation to the cooler model, which is the target point. Figure 8.4 shows the

mobile robot manipulator with frames labeled at all joints, from the MiR base to the camera, and the tip of the drill. These frames are used to calculate the motion of the mobile robot manipulator.

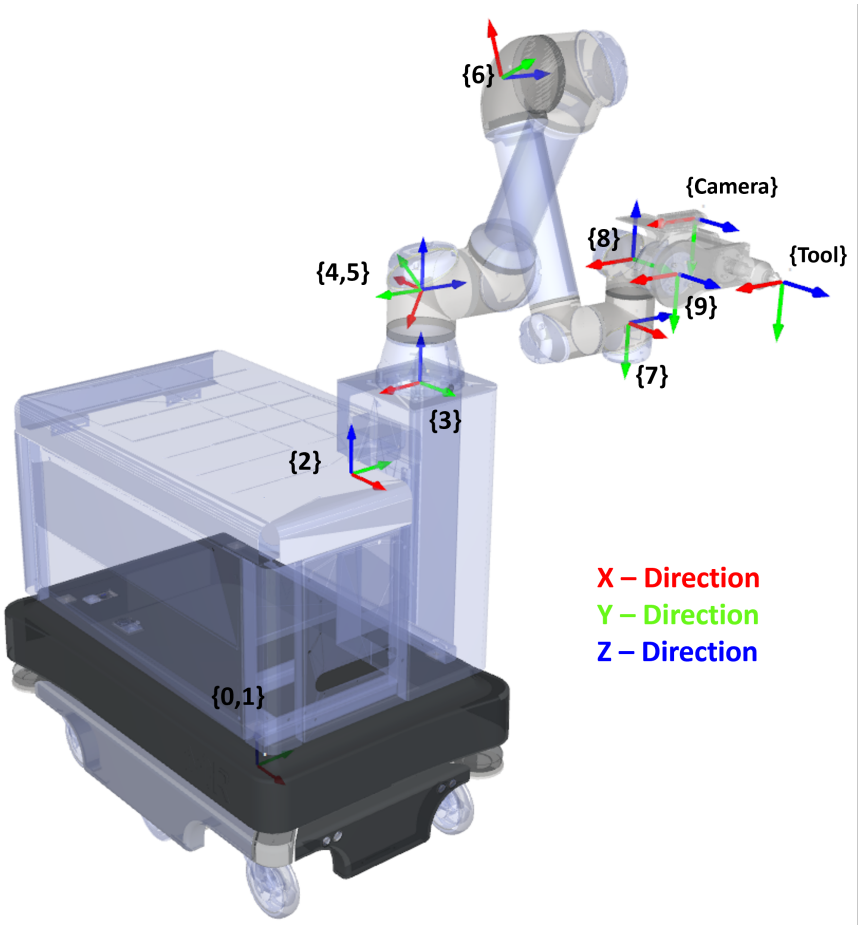


Figure 8.4: Mobile robot manipulator with associated Frames

MiR200 Control

Controlling the MiR200 mobile robot is a key aspect of the automation process. The primary role of the MiR200 is to transport the UR5e manipulator to a position in front of the cooler, at the optimal distance that allows the UR5e to assume its ideal configuration for the chamfering process. This task is accomplished through a Python script that controls the movements of the MiR200, and can be found in Appendix J.

A Python script, which serves as one of the modules for automating the tilting process, is designed to control the MiR200 by publishing speed commands (Twist messages) to the `/cmd_vel` topic and subscribing to the robot's odometry messages from the `/odom` topic.

The script defines a MiR200Controller class that initializes the ROS node, sets the publisher and subscriber, and defines methods to move the robot a certain distance and rotate it by a certain angle. The `move_distance` method publishes Twist mes-

sages to command the robot to move a specified distance at a specified speed. The rotation method does the same but for rotation.

The script also includes methods to calculate distance and angle differences, which are used to determine when the robot has moved or rotated the desired amount. The current position of the robot is updated in the `odom_callback` method, which is called every time a new odometry message is received.

The script is designed to move the robot to a specific position in front of the cooler, as defined by the target frame. This frame is located in front of the cooler and has the desired orientation that the robot must achieve. This implies that the base frame of the MiR200 must be aligned with the target frame, as shown in Figure 8.5. The robot movements are controlled to reach the target frame, ensuring that the UR5e manipulator is correctly positioned for the chamfering process.

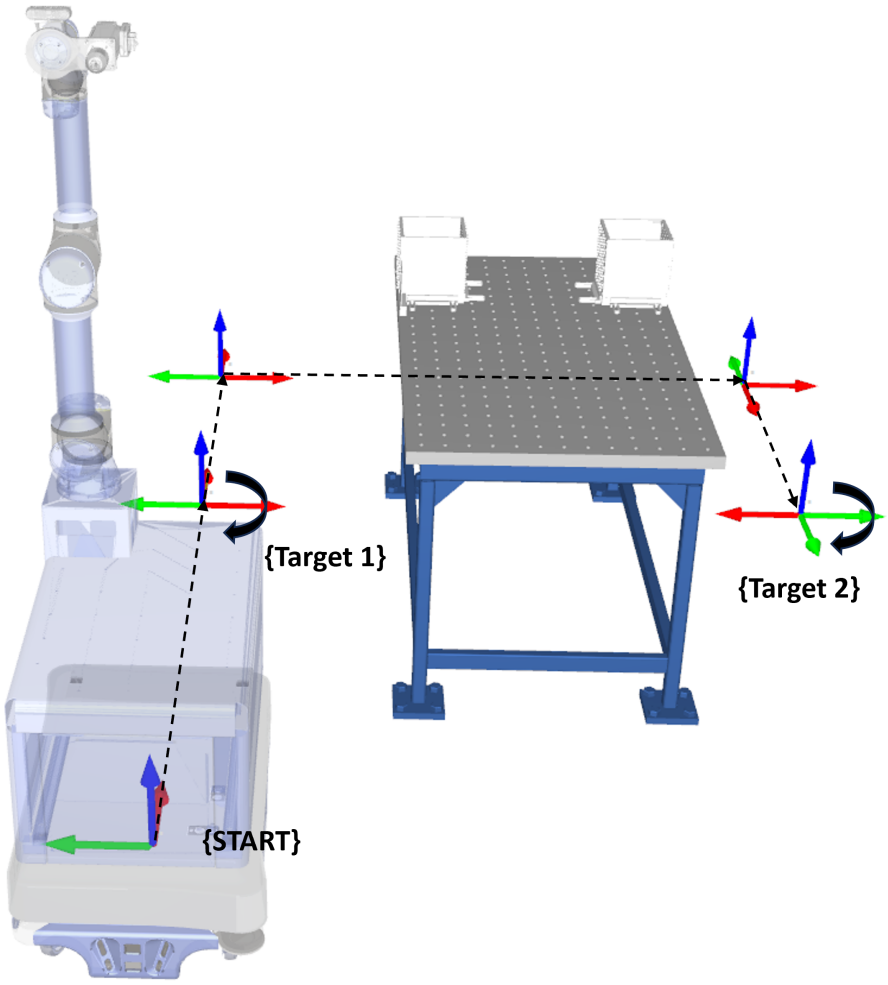


Figure 8.5: MiR200 movement Plan

The movement of the MiR200 can be mathematically represented. If we denote the initial position of the MiR200 in the world frame as (X_0, Y_0) and the target position as (X_t, Y_t), the displacement in the X and Y directions can be represented as:

$$\Delta X = X_t - X_0 \tag{8.3}$$

$$\Delta Y = Y_t - Y_0 \tag{8.4}$$

The MiR200 needs to rotate by an angle $\theta = atan2(\Delta Y, \Delta X)$ to align itself with the target position. After rotating by θ, the MiR200 can move directly towards the target position.

Given that the workspace for performing this process is prepared in advance, the initial position of the mobile robot manipulator is known in advance, as well as the position of the cooler in relation to it. Based on the Equations 8.3 and 8.4 movements of MIR200 were calculated in order to reach the desired Target 1. Furthermore, the robot also has a defined target in front of the second cooler and the path to reach Target 2, which is defined in a similar way, as well as the return to the home target after the completion of the procedure. The whole plan is shown in Figure 8.5.

UR5e Control

Controlling the UR5e manipulator is probably the most important part of the automation process because it performs the process that the goal is to automate, which is the chamfering process. After the MiR200 mobile robot reaches the desired target position, the UR5e manipulator takes the optimal configuration for chamfering, as calculated in Section 6. Then the end effector alignment procedure is performed, by setting the end effector in the same plane as the cooler, based on the alignment with the checkerboard frame that the camera recognized. This ensures that the end effector is parallel to the cooler, and ensures precision in the process. After that, the camera detects the pipes, and based on the recognized positions of the pipes, the chamfering process starts. Camera recognition procedures are explained in detail in Chapter 7. This process is a key module in the planned automation process, which is explained in Section 3.2.

The chamfering process is managed by a Python script, which can be found in Appendix I. The script begins by initializing the `moveit_commander` and the UR5e group. It then sets the reference frame (base) for the robot and the maximum velocity and acceleration scaling factors. The script reads the pipe coordinates from a text file, a critical step as the entire process is based on these coordinates.

The script then calculates the transformation matrix from the end effector to the base link frame using the `tf` package. For each pipe center, the script applies a transformation from the camera to the end-effector to obtain the position of the pipe center within the end-effector frame. Given that the drill is rigidly attached to the end-effector, the translational component of the transformation from the drill tip to the end effector is subtracted from the position of the pipe in the end-effector frame. This operation yields the position of the drill tip in the pipe centers. This transformation is then converted into the base frame, as the robot's positioning is

performed by defining the transformation of the end-effector relative to the base frame.

The script then moves the drill tip to the center of each pipe, with an offset in depth by a predefined value. It first retrieves the current pose of the end-effector in the base frame and converts it into a homogeneous transformation matrix. It then computes the inverse transformation matrix from the base to the end-effector and transforms the current pose from the base frame to the end-effector frame. Within the end-effector frame, it modifies the Z coordinate by the amount it intends to move. It then transforms the new pose from the end-effector frame back to the base frame and converts the new pose from a homogeneous transformation matrix into a Pose message. The script then reduces the robot's speed to perform the drilling operation and moves the end-effector to a new position at a predefined depth where drilling takes place.

After drilling, the end-effector moves linearly back by the same amount, and the robot's speed is increased to move back and position itself in front of the next pipe. These actions are repeated until the chamfering of all pipes, as defined in the loaded text document, is completed.

The chamfering process involves a series of transformations to accurately position the drill tip at the center of each pipe. The transformations are calculated as follows:

1. The transformation matrix from the end effector to the camera, obtained by Eye-in-hand calibration (Section 7.2), denoted as $T_{\text{ee\_to\_camera}}$, and the transformation matrix from the end-effector to the drill, obtained by measuring offsets, denoted as $T_{\text{ee\_to\_drill}}$, are defined as:

$$T_{\text{ee\_to\_camera}} = \begin{bmatrix} 0.99703091 & -0.05133999 & -0.05738971 & -0.02912 \\ 0.05427406 & 0.99723339 & 0.05079257 & -0.07788 \\ 0.05462325 & -0.05375654 & 0.99705894 & -0.00418 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{\text{ee\_to\_drill}} = \begin{bmatrix} 1 & 0 & 0 & -0.0973 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.129 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. The transformation matrix from the base to the end-effector, denoted as $T_{\text{base\_to\_ee}}$, is calculated using the tf package. This transformation is the result of a series of transformations from the base to the end effector through each joint:

$$T_{\text{base\_to\_ee}} = T_{\text{base\_to\_shoulder}} \times \ldots \times T_{\text{wrist\_2\_to\_wrist\_3}} \tag{8.5}$$

3. For each pipe center, the script applies the transformation from the end-effector to the camera to obtain the position of the pipe center within the end-effector frame:

$$p_{\text{pipe\_in\_ee}} = T_{\text{ee\_to\_camera}} \times \text{pipe\_coords} \tag{8.6}$$

4. The translational component of the transformation from the end-effector to the drill is subtracted from the position of the pipe in the end-effector frame to obtain the position of the drill tip in the pipe centers:

$$T_{\text{pipe\_in\_ee}} = T_{\text{pipe\_in\_ee}} - T_{\text{ee\_to\_drill}} \tag{8.7}$$

5. This transformation is then converted into the base frame:

$$T_{\text{pipe\_in\_base}} = T_{\text{base\_to\_ee}} \times T_{\text{pipe\_in\_ee}} \tag{8.8}$$

6. The script then moves the tip of the tool to the center of each pipe, offset in depth by a defined value. It first obtains the current position of the end-effector in the base frame and converts it into a homogeneous transformation matrix:

$$T_{\text{base\_to\_current}} = \texttt{get\_current\_pose\_in\_frame(group, 'base')} \tag{8.9}$$

7. It then computes the inverse transformation matrix from the base to the end-effector, denoted as $T^{-1}_{\text{ee\_to\_base}}$, and transforms the current pose from the base frame to the end-effector frame:

$$T_{\text{ee\_to\_current}} = T^{-1}_{\text{ee\_to\_base}} \times T_{\text{base\_to\_current}} \tag{8.10}$$

8. Within the end-effector frame, it modifies the Z coordinate by the amount it intends to move:

$$T_{\text{ee\_to\_current}}[2,3] += z_{\text{offset}} \tag{8.11}$$

9. It then transforms the new pose from the end-effector frame back to the base frame:

$$T_{\text{base\_to\_new}} = T_{\text{ee\_to\_base}} \times T_{\text{ee\_to\_current}} \tag{8.12}$$

This sequence of transformations allows the script to accurately position the drill tip at the center of each pipe, offset in depth by a predefined value, and perform the chamfering operation.

The whole process is illustrated in Figure 8.6, where the different frames are displayed, including the joint frames of UR5e, as well as frames of the camera and tip of the drill, and target frames on pipes.

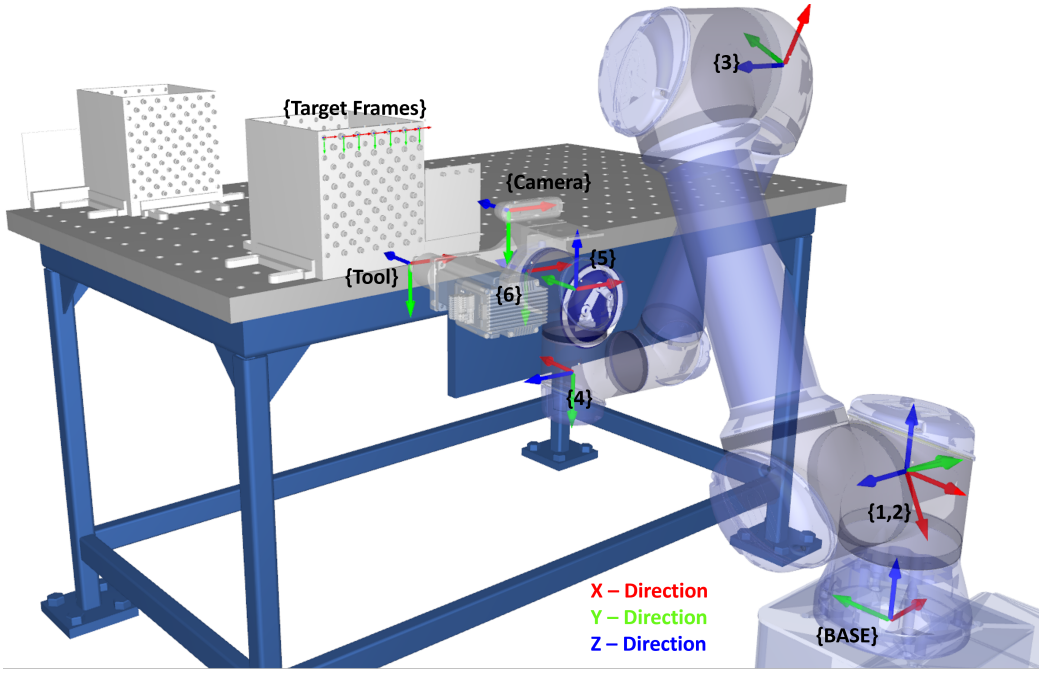


Figure 8.6: UR5e movement plan for Chamfering Process

8.4 Results and validation

Regarding the results of the testing procedure, it is evident that the procedure is not flawless. Several factors contribute to this outcome. Firstly, the decision to use real space coordinates obtained from the camera, as opposed to the mean value for the Z plane, introduces variations in drilling depth across the pipes. Consequently, the initial rows experienced significant deviations. Adjusting the offset resulted in the drill not reaching an adequate depth for capturing the chamfering process. Conversely, the procedure exhibited promising outcomes on the opposite side of the cooler, where most pipes were accurately recognized, and the chamfering distance was correct, leading to successfully chamfered plates.

It is important to acknowledge that these results do not meet industrial standards. However, the project group firmly believes that with further optimization efforts and dedicated full-time engineering resources, the procedure can be refined to perfection. The primary objective here was to demonstrate the feasibility of the procedure, even with limited resources.

Regarding the offset in chamfering, several potential explanations arise. One possibility is an error in the camera eye-in-hand calibration results. Even if the initial results were accurate, subsequent misalignment could have occurred, resulting in an incorrect transformation matrix. Additionally, rounding errors in calculating the transformation from the base to the target point (pipe) could contribute to the offset. The movement of the cooler on the table, as depicted in the video where one tester holds the cooler, could also introduce variations. Furthermore, the robot's movement during the chamfering procedure or any unintended base movement could lead to incorrect target points. Also, it is known that camera can give different results when

reading depth. These factors should be thoroughly investigated as part of future work, which constitutes a key recommendation.

In conclusion, the "Robot Programming and Movement" chapter highlights the obtained results and acknowledges areas for improvement. The demonstrated procedure, although imperfect, proves the feasibility of automation even with limited-budget equipment. The identified limitations and recommendations pave the way for further optimization and research in the future. The results are best observed in the video which can be accessed from the code in Figure 8.8, but the plates can also be seen in Figure 8.7.

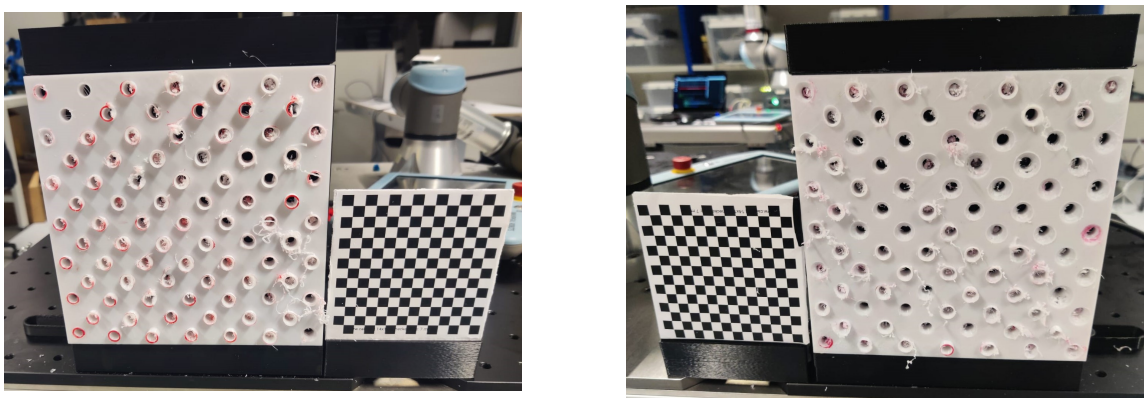


Figure 8.7: Results of the procedure test

Figure 8.8: QR Code to access the video [`https://youtu.be/zeCs1SegMWQ`]

8.5 Summary

The Chapter provides a detailed overview of the control mechanisms and programming used for the UR5e manipulator and the MiR200 mobile robot. It begins by exploring MoveIt Commander, the key tool for planning the motion of the UR5e robot in ROS, and explains the role of publishers and subscribers in the ROS system.

The specifications of controlling the MiR200 mobile robot are also considered, including the use of the odometry callback function to understand the position and orientation of the robot and methods of moving the robot to a certain distance and its rotation.

The calculations involved in determining the robot's motion are covered and the functionality of the controllers is tested. The final part discusses the integration of UR5e and MiR200 into a mobile robotic manipulator system. This includes creating a MoveIt package for the system, establishing system communication, and operating a mobile robot manipulator system to automate the chamfering process.

The results obtained in the "Robot Programming and Movement" chapter are effectively captured in the created video of the final procedure. This video showcases the testing conducted at the University, specifically on the cooler manufactured using 3D printers. The procedure executed in the video aligns with the depicted flow chart in Section 3.2.

The primary objective of this chapter was to demonstrate that each module can be successfully executed and that it is feasible to integrate all the modules into a synergistic system. The testing procedure aimed to validate this concept and provide tangible evidence of its viability.

9 | Digital Twin Development

9.1 Introduction

The concept of a Digital Twin has gained significant prominence in the field of automation and robotics. It refers to a virtual replica of a physical system or process, allowing for real-time monitoring, simulation, and analysis. Digital Twins are utilized for a variety of purposes, including performance optimization, predictive maintenance, and process improvement. In this chapter, the development and implementation of a Digital Twin for this automation project are explored, focusing on mimicking real robot movements and monitoring joint torques.

9.2 Methodology

The primary goal of Digital Twin is to closely replicate the movements and behavior of the real robot, specifically the UR5e robot and its joint torques during the chamfering process. The Digital Twin is built upon the Gazebo simulation environment, which provides a physics-based virtual environment to simulate and visualize the robot's actions. Alongside the simulation in the Gazebo, the torque plot for each joint will be visible. After the procedure, the text file will be saved with the positions of the joints alongside the torques. The Digital Twin is created on a separate computer, wirelessly connected to the main system comprising the robot and end-effector. This wireless connection is established through the Robot Operating System (ROS) master running on the main laptop. The ROS master facilitates seamless communication and data exchange between the real system and the Digital Twin, enabling real-time synchronization and monitoring.

To offload the simulation in Gazebo to a second laptop, a distributed system is set up where processing is divided. General procedure that was followed.

- Set up network communication: both machines should be connected to the same network, either via Ethernet cable or Wi-fi
- ROS & Gazebo should be installed on both machines
- Configure ROS Master: one laptop should act as a ROS Master, that laptop will coordinate the communication between the machines. In this case, it is natural to choose the first laptop that starts the `.launch` file to serve as a Master. On both machines the `ROS_MASTER_URI` environment variable should be set as:

  ```
  export ROS_MASTER_URI=http://192.168.12.250:11311
  ```

  Wher IP address: `192.168.12.250` is the IP address of the first laptop.
- Next, `ROS_IP` environment variable should be set as:

```
export ROS\_IP=192.168.12.250  # On first laptop
export ROS\_IP=192.168.12.248  # On second machine
```

- Once everything is set, the main .launch file can be started, which loads the drivers
- Finally, the Gazebo simulation package can be started on the second laptop, together with joint torque live plot

It is important to note that each new terminal that is initialized on the second laptop should first set up `ROS_MASTER_URI` and `ROS_IP` environment variable. Since both machines need full bi-directional connectivity, on all ports - if there is a problem with communication few steps for debugging that helped during the project are:

- Check the internet connection on both machines
- Try to ping each machine from itself as for example machine 2:

```
ssh 192.168.12.248
ping 192.168.12.248
```

- Try to ping from opposite machines as for example:

```
ssh 192.168.12.248
ping 192.168.12.250
```

- Try to check for `rostopic list`, if same `rostopics` are present in both machines connection is successful
- Lastly, since the Gazebo simulation is built around `\joint _states` message that publishes information about joints both from MiR200 and UR5e about position, velocity, and effort, try to `rostopic echo \joint_states` if the message is empty - there is a problem with a publisher

To visualize the robot in the Gazebo on the second laptop, a new package called `gazebo_digital_twin` was created. The package incorporates the necessary launch files and code to enable communication and control between the real robot and the Digital Twin.

The launch file within the `gazebo_digital_twin` package plays a crucial role in initializing and configuring the Digital Twin. The URDF file that was created before stays the same, since the description of the robot did not change. Here is an overview of the launch file that can be seen in Appendix D.1:

- The URDF file, `ur5e_mir200.xacro`, is loaded into the parameter server using the `robot_description` parameter.
- The Gazebo world is started using the `empty_world.launch` file from the `gazebo_ros` package.
- The robot is spawned in Gazebo with the specified URDF description.

- Joint controller configurations are loaded from a YAML file into the parameter server.
- The controllers for the UR5e arm and MiR200 base are loaded using the `controller_manager` package.
- The `joint_state_replicator` node replicates the joint states from the real robot in Gazebo.
- The `cmd_vel_forwarder` node forwards `cmd_vel` messages to the mobile base controller.

The `joint_state_replicator` script within the `gazebo_digital_twin` package replicates the joint states of the real UR5e robot in Gazebo. Here is an overview of the code that can be found in Appendix D.2:

- The node is initialized as `robot_controller`.
- The `joint_state_callback` function is called when joint states are received from the real robot.
- The joint states for the UR5e robot are extracted and transformed into a `JointTrajectory` message.
- The `JointTrajectory` message is published to the `/ur5e_arm_controller/command` topic, replicating the joint movements in Gazebo.

And lastly, The `cmd_vel_forwarder` script in Appendix D.3, forwards the `cmd_vel` messages received to the mobile base controller.

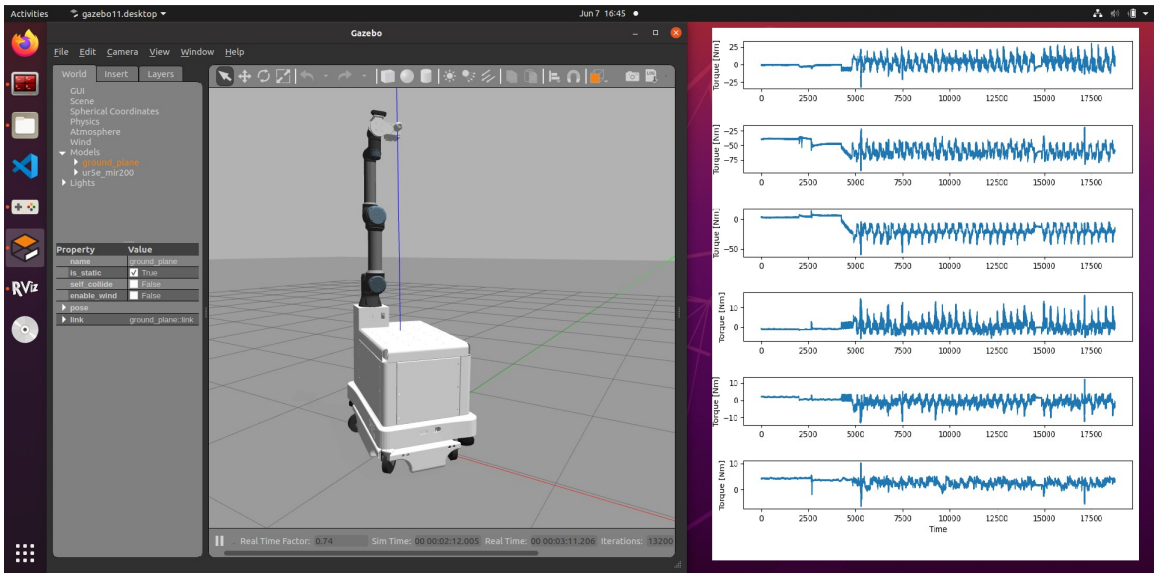


Figure 9.1: Digital Twin - end of process (Gazebo and Joint torques plot)

The final result of the Digital Twin can be seen in Figure 9.1, where after chamfering

one side of the cooler the state of the robot together with the torque plot was captured.

9.3 Benefits

The Digital Twin provides the capability to analyze joint torques during and after the chamfering procedure. By monitoring the torques exerted on each joint, insights can be gained regarding the joints that experience the highest forces during chamfering. This information can guide adjustments in the robot's configuration to optimize the procedure and minimize stress on specific joints.

One significant advantage of the Digital Twin is the improved safety it offers. By allowing the user or worker to observe the procedure from a safe distance, potential risks associated with the robot's high-speed movements and the spinning drill are mitigated. This enhanced safety is achieved by enabling remote control and monitoring of the chamfering process through the Digital Twin.

9.4 Post-processing and optimization

With the Digital Twin, the entire chamfering procedure can be timed precisely. By accurately measuring the time taken by the real system and comparing it with the Digital Twin, the efficiency and performance of the process can be assessed. This information serves as a foundation for procedure optimization, allowing for adjustments such as increasing the speed of the MiR200 without compromising accuracy.

The Digital Twin enables post-processing and analysis of the collected torque and position data. By saving the torques to a text file, they can be further analyzed and combined with position information. For example, observing a small overshoot of the end-effector during transitions between pipes can lead to fine-tuning of the movement parameters, such as testing different configurations (elbow up or down) or adjusting the distance between the end-effector and the MiR200. These refinements help improve the overall stability and precision of the chamfering process.

Digital signal processing (DSP) is the mathematical manipulation of an information signal, such as audio, for the purpose of enhancing or improving the quality of the original signal [68]. One common use of DSP is filtering, which is the selective removal of certain aspects of the signal, in order to emphasize other aspects that are considered important.

One such type of filter used in DSP is the Butterworth filter. The Butterworth filter, also known as a maximally flat magnitude filter, is characterized by a frequency response that is flat as possible in the passband and has an adequate roll-off rate in the stopband [69]. It is designed to have the optimal trade-off between the passband and stopband performance, thus maximizing the bandwidth for a given degree of approximation.

In this work, a low-pass Butterworth filter is employed. The low pass filter allows signals with a frequency lower than a certain cutoff frequency to pass through and

attenuates signals with frequencies higher than the cutoff frequency. The degree of the filter is often called the order of the filter. Here, a second-order Butterworth filter is used. The order of a filter is an important characteristic that determines the filter's complexity and its sharpness in the transition between the passband and the stopband. A higher-order filter would have a sharper transition, but it also requires more computational resources.

The Butterworth filter is defined by a transfer function $H(s)$ [70]:

$$H(s) = \frac{1}{\sqrt{1 + (\frac{s}{\omega_c})^{2n}}} \tag{9.1}$$

Where:

- s is the complex frequency
- ω_c is the cutoff frequency
- n is the order of the filter

In the case of a second-order low-pass Butterworth filter with a cutoff frequency of $\omega_c = 1$, the equation for the frequency response is as follows:

$$H(s) = \frac{1}{\sqrt{1 + (\frac{s}{1})^{4}}} \tag{9.2}$$

Figure 9.2 illustrates a comparison between the original joint torque signals and the filtered signals in the time domain. By inspecting this figure, it is evident that the low-pass Butterworth filter has effectively attenuated the high-frequency noise in the joint torques.

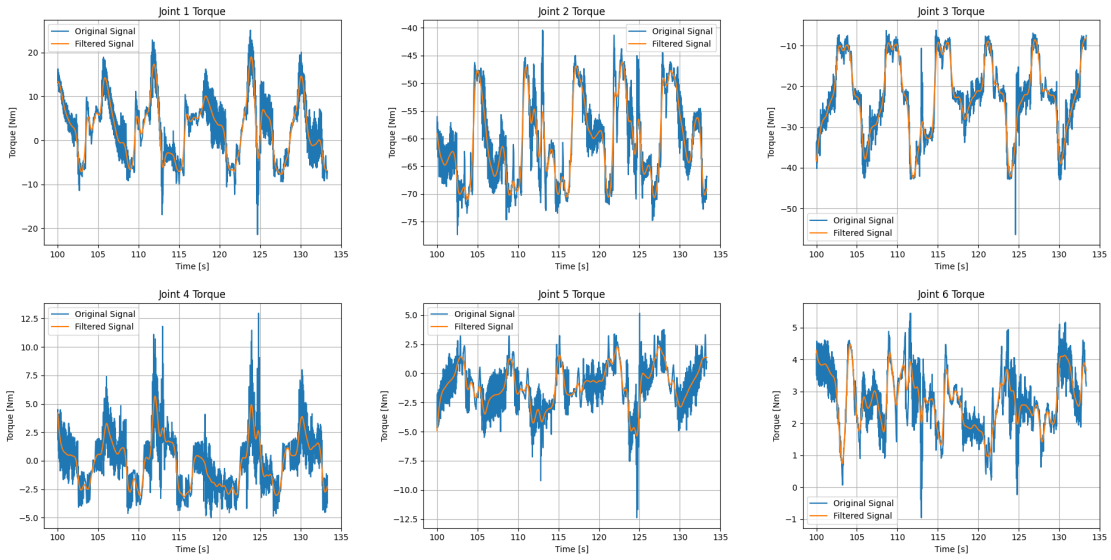


Figure 9.2:

Figure 9.3 demonstrates the one-sided amplitude spectrum of the joint torques. The cutoff frequency was chosen based on this figure. The cutoff frequency of 1 Hz was chosen to preserve the high amplitude signals while effectively attenuating the noise signals with higher frequencies.

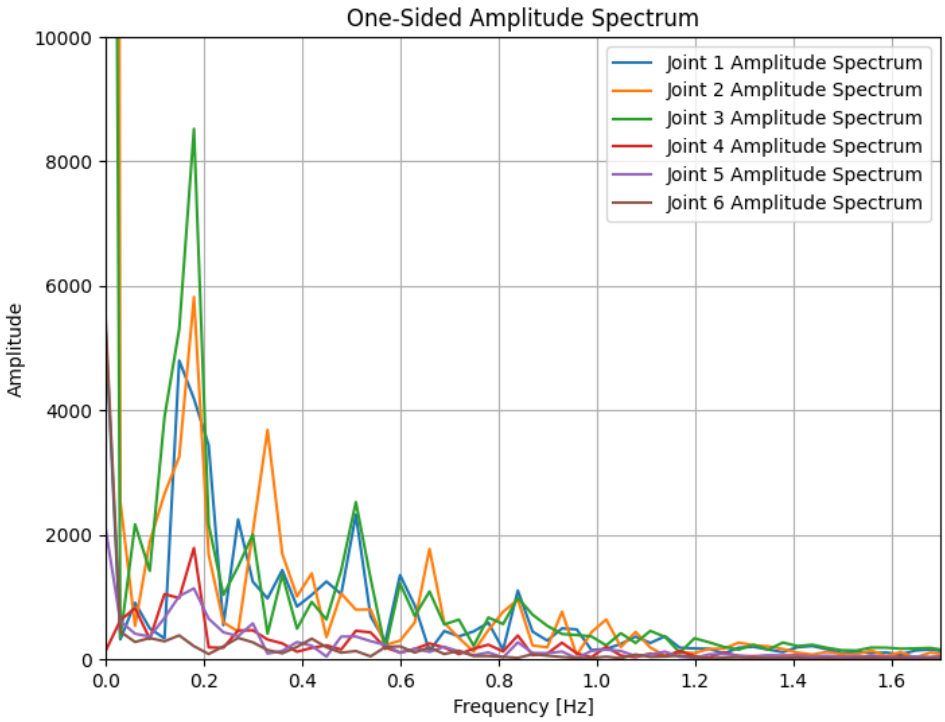


Figure 9.3: One sided amplitude spectrum

The frequency response of a filter describes how the magnitude and phase of the output signal change with respect to the input signal frequency. Figure 9.4 illustrates the frequency response of the applied Butterworth filter. This plot is important as it gives an insight into how different frequency components of the signal will be attenuated by the filter. From this plot, it is possible to visually assess the passband, stopband, and roll-off rate of the filter.

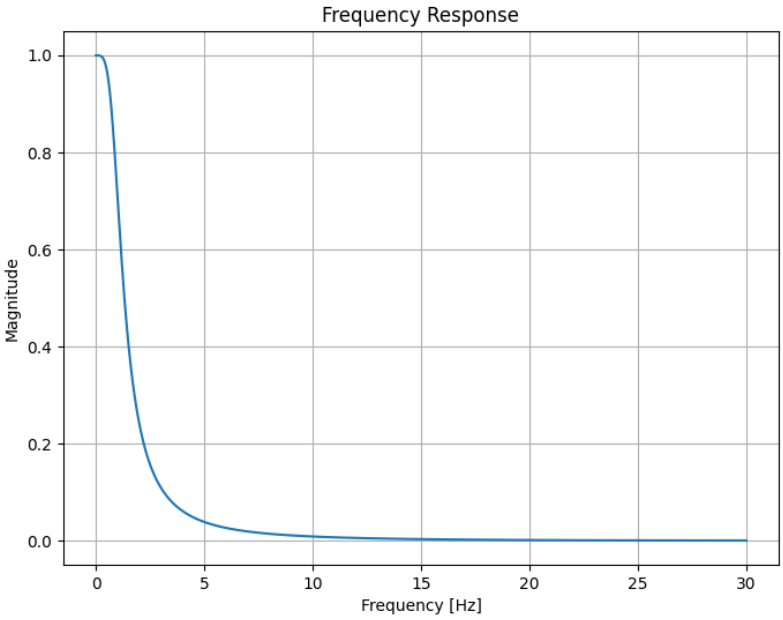


Figure 9.4: Frequency response

The procedure of signal processing was done in Python using `scipy.signal` library and can be seen in Appendix E.

The final diagram representing connections between nodes and topics can be seen in Figure 9.5. The new nodes `n_gazebo, n_cmd_vel_forwarder, n_joint_state_replicator` painted in green represent added nodes after implementing Digital Twin.

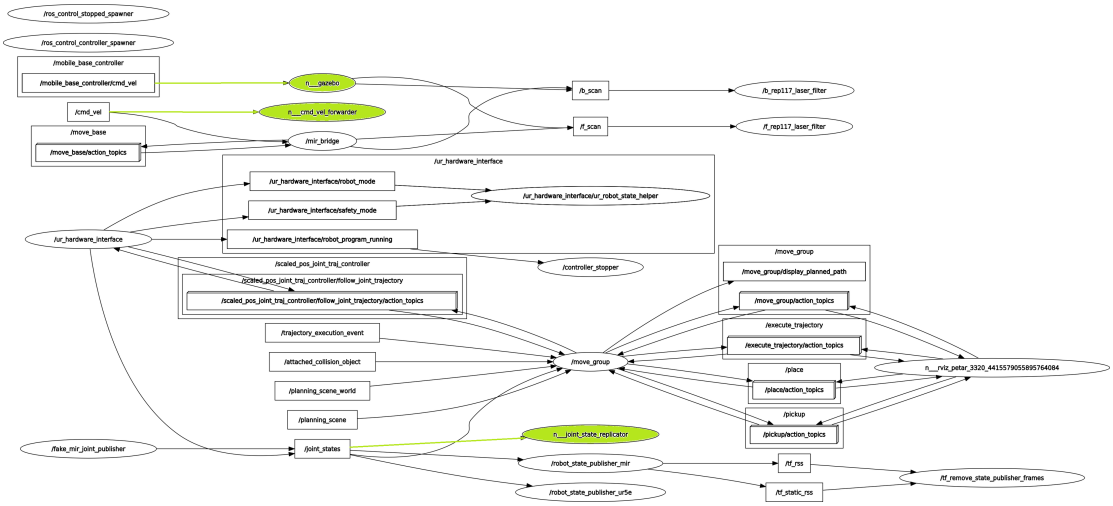


Figure 9.5:

9.5 Summary

The Digital Twin, built upon the Gazebo simulation environment, offers valuable capabilities for mimicking real robot movements, monitoring joint torques, enhancing safety, and optimizing the chamfering procedure. By leveraging wireless network connectivity, precise timing, and post-processing analysis, the Digital Twin serves as a powerful tool for observation, analysis, and refinement of the automation process.

10 | Conclusion & Discussion

10.1 Summary of the project and its outcomes

Presented methods and experiments with the usage of real robots and designed components provided prototypes, which proved the idea and topic of the thesis. Under further development, the outcome of the described work could be used for industrial purposes, including the improvement of the existing chamfering process of pipes for cooler manufacturers to make it autonomous. Provided data and developed models can be a concrete platform for the future development of the autonomous chamfering process of pipes. The most important results from the project are actually the results of testing the final procedure from Section 8.4, and most of the conclusions made there, can be translated as final conclusions of the projects.

10.2 Significance of the results for process automation and industrial robotics

Providing a study on the automation of the project could provide multiple advantages including a reduction of the production time, as well as provide more accurate chamfering with the robust model. Additionally, once purchased equipment can substitute human labor and reduce the risk of mistakes and accidents during the work.

10.3 Recommendations for future work

Further improvement of explicit dynamic simulation with mesh and mesh-less could be adequate for future work to increase the accuracy of simulated results, as well as include thermal effects during the chamfering process on pipes. Moreover, proven SPH simulation was created based on the slower response of the drill, adjustment of mesh with more elements and better computer power could provide much better results. Moreover, the theoretical linear model that was developed for forecasting forces and moments occurring during the drilling or chamfering is very simplified and might not fit in most applications as well for multiple types of materials, therefore much better one should be developed and investigated based on existing studies.

One of the main points in the camera vision topic would be the effort to increase accuracy for detecting pipes, as well as recognizing depth. One of the ways to increase accuracy would be the usage of better equipment as well as additional sensors for depth detection. Moreover, the YOLO model could be further improved and its database updated with new cooler types to make detection more robust and stable.

As was mentioned in previous studies [22] to assure that the robot arm can conduct chamfering process, an adequate robot arm for industrial purposes should be chosen,

as well as a mobile robot. This improvement could provide better resources and a faster process with a lower chance of failure.

To further enhance the capabilities and performance of the digital twin system, the following recommendations are proposed:

Create Ideal Environmental Conditions: It is recommended to establish and maintain ideal environmental conditions within the digital twin setup in Gazebo. This includes ensuring proper lighting, temperature, and other relevant factors that closely resemble real-world operating conditions.

Validate Controller Performance: To improve the fidelity of the digital twin, it is advisable to conduct comprehensive testing and validation of the controllers used in the digital twin. A comparative analysis with the controllers employed in the real robot can be performed to identify any discrepancies and optimize the controller parameters.

Utilize a Dedicated Mini PC: Instead of relying on a laptop for the digital twin setup, integrating a dedicated mini PC within the robot's casing or housing is recommended. This solution offers better protection for the computational hardware and eliminates the reliance on external devices that may be susceptible to damage.

Furthermore, the implementation of Real-Time Position Optimization rather than pre-calculating the optimal position before chamfering or for each pipe would be a good idea, where a dynamic approach can be adopted. By implementing a live function within the process, the system can continuously evaluate and calculate the optimal position for each pipe based on real-time inputs.

Bibliography

[1] J. E. Dølvik and J. R. Steen, "Automation, workers' skills and job satisfaction." https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0242929, 2020.

[2] D. Acemoglu and P. Restrepo, "Artificial intelligence, automation and work." https://www.nber.org/papers/w24196, 2018.

[3] D. Acemoglu and P. Restrepo, "Robots and jobs: Evidence from us labor markets," 2023.

[4] E. Brynjolfsson and T. Mitchell, "What can machine learning do? workforce implications," 2023.

[5] M. Chui, J. Manyika, and M. Miremadi, "Where machines could replace humans—and where they can't (yet)," 2023.

[6] S. Garnier, K. Subrin, and K. Waiyagan, "Modelling of robotic drilling," *Procedia CIRP*, vol. 58, pp. 416–421, 12 2017.

[7] V. P. S. J.R.V. Sai Kiran, "Simulation and experimental research on robot drilling," *International Journal of Recent Technology and Engineering (IJRTE)*, 2019.

[8] G. C. B. B. Pereira, B., "Optimization of an autonomous robotic drilling system for the machining of aluminum aerospace alloys," *Procedia CIRP*, 11 2021.

[9] J. Zhang, W. Liao, Y. Bu, W. Tian, and J. Hu, "Stiffness properties analysis and enhancement in robotic drilling application," *The International Journal of Advanced Manufacturing Technology*, vol. 106, 02 2020.

[10] V. Gyliene, "Drilling process modelling using sph." https://www.dynalook.com/conferences/9th-european-ls-dyna-conference/drilling-process-modelling-using-sph, 2013.

[11] W.-d. Zhu, B. Mei, G.-r. Yan, and Y.-l. Ke, "Development of a monocular vision system for robotic drilling," *Journal of Zhejiang University SCIENCE C*, vol. 15, no. 8, pp. 593–606, 2014.

[12] A. Ayyad, M. Halwani, D. Swart, R. Muthusamy, F. Almaskari, and Y. Zweiri, "Neuromorphic vision based control for the precise positioning of robotic drilling systems," *Robotics and Computer-Integrated Manufacturing*, vol. 79, p. 102419, 2023.

[13] S. Ji, T. Saravirta, S. Pan, G. Long, and A. Walid, "Emerging trends in federated learning: From model fusion to federated x learning," *arXiv preprint arXiv:2102.12920*, 2021.

[14] Y. Chen, A. Goodridge, M. Sahu, A. Kishore, S. Vafaee, H. Mohan, K. Sapozhnikov, F. X. Creighton, R. H. Taylor, and D. Galaiya, "A

force-sensing surgical drill for real-time force feedback in robotic mastoidectomy," *International Journal of Computer Assisted Radiology and Surgery*, pp. 1–8, 2023.

[15] Universal Robots, "Universal robots - ur5e robot." `https://www.universal-robots.com/da/produkter/ur5-robot/`, N.D. Accessed: May 24, 2023.

[16] Mobile Industrial Robots, "Mir200." `https://www.konicaminolta.com.au/products/robotics/mir200`. Accessed: June 5, 2023.

[17] i2r, "Er-ability." `https://www.i2r.dk/robotbutikken/mobile-robotter-og-tilbehoer/mobile-platforme-med-robot/er-ability/`. Accessed: June 5, 2023.

[18] R. Wiki, "Urdf." `http://wiki.ros.org/urdf`, 2021.

[19] M. Mayer, J. Külz, and M. Althoff, "Cobra: A composable benchmark for robotics applications," 2022.

[20] D. Huczala, T. Kot, J. Mlotek, J. Suder, and M. Pfurner, "An automated conversion between selected robot kinematic representations." `http://arxiv.org/abs/2204.02629v2`, 2022.

[21] R. Wiki, "Moveit!." `http://wiki.ros.org/moveit`, 2021.

[22] A. K. Adam Mariusz Kuryła, Petar Gavran, "Preliminary automatization of chamfering process using robot technology on charge air cooler/egr for vestas aircoil a/s," 2022.

[23] A. T. . A. L. Svensson, D., *A. Coupled Eulerian–Lagrangian simulation and experimental investigation of indexable drilling.* PhD thesis, 2022.

[24] A. Harish, "Implicit vs explicit finite element method (fem)." `https://www.simscale.com/blog/implicit-vs-explicit-fem/`, 2023.

[25] T. Ltd, "What is pla? (everything you need to know)," 2023.

[26] Ansys, "Ansys ls-dyna." `https://www.ansys.com/products/structures/ansys-ls-dyna`, 2023.

[27] Wikipedia, "High-speed steel — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=High-speed_steel&oldid=1107574202`, 2022. Accessed: 4-June-2023.

[28] Dynamore, "Ls-prepost." `https://www.dynamore.de/en/products/pre-and-postprocessors/prepost`, 2023.

[29] S. Mitchell, "Choosing corners of rectangles for mapped meshing.," pp. 87–93, 01 1997.

[30] LS-Dyna, "Mass scaling." `https://www.dynasupport.com/howtos/general/mass-scaling`, 2023.

[31] G. A. Şenol Şirin, Enes Aslan, "Effects of 3d-printed pla material with different filling densities on coefficient of friction performance." https://www.emerald.com/insight/content/doi/10.1108/RPJ-03-2022-0081/full/html, 2022.

[32] N.-H. Chu, D.-B. Nguyen, N.-K. Ngo, V.-D. Nguyen, M.-D. Tran, N.-P. Vu, Q.-H. Ngo, and T.-H. Tran, "A new approach to modelling the drilling torque in conventional and ultrasonic assisted deep-hole drilling processes." https://www.mdpi.com/2076-3417/8/12/2600, 2018.

[33] E. I. Shchurova, "Radial force calculation at the start of drilling operation using the spg method," in *Proceedings of the 7th International Conference on Industrial Engineering (ICIE 2021)* (A. A. Radionov and V. R. Gasiyarov, eds.), (Cham), pp. 119–128, Springer International Publishing, 2022.

[34] MathWorks, "Robotics system toolbox - matlab." https://www.mathworks.com/products/robotics.html, 2021.

[35] MathWorks, "fmincon - matlab." https://www.mathworks.com/help/optim/ug/fmincon.html, 2021.

[36] R. Tsai and R. Lenz, "A new technique for fully autonomous and efficient 3d robotics hand/eye calibration," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 3, pp. 345–358, 1989.

[37] Zivid, "How to use the result of hand-eye calibration." https://support.zivid.com/en/latest/academy/applications/hand-eye/how-to-use-the-result-of-hand-eye-calibration.html, N.D. Accessed: May 24, 2023.

[38] K. Daniilidis, "Hand-eye calibration using dual quaternions," *The International Journal of Robotics Research*, vol. 18, no. 3, pp. 286–298, 1999.

[39] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[40] G. Bradski, *Open Source Computer Vision*. O'Reilly Media, 2008.

[41] OpenCV, "Opencv: Open source computer vision library." https://github.com/opencv/opencv, 2023.

[42] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[43] K. Madsen, H. Nielsen, and O. Tingleff, "Methods for non-linear least squares problems," pp. 24–29, 2004.

[44] F. Devernay and O. Faugeras, "Straight lines have to be straight," *Machine Vision and Applications*, vol. 13, pp. 14–24, July 2001.

[45] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[46] Wikipedia, "Rodrigues' rotation formula." https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula, 2023.

[47] A. G. Valdenebro, "Visualizing rotations and composition of rotations with rodrigues' vector," *European Journal of Physics*, vol. 37, no. 6, p. 065001, 2016.

[48] Open Source Robotics Foundation, *tf: ROS transform library*, 2020.

[49] G. H. Golub and C. F. Van Loan, *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 2012.

[50] H. Germain, V. Lepetit, and G. Bourmaud, "Neural reprojection error: Merging feature learning and camera pose estimation," 2021.

[51] Doxygen, "Hough circle transform." https://docs.opencv.org/4.x/da/d53/tutorial_py_houghcircles.html, 2023.

[52] B. V. LLC, "Blob detection." https://learnopencv.com/blob-detection-using-opencv-python-c/, 2023.

[53] Ultralytics, "Yolo algorithm." https://ultralytics.com/yolov5, 2023.

[54] P. Lihi Gur Arie, "Training of model for yolo." https://towardsdatascience.com/the-practical-guide-for-object-detection-with-yolov5-algorithm-74c04aac4843, 2022.

[55] P. Baheti, "Train test validation split." https://www.v7labs.com/blog/train-validation-test-set, 2021.

[56] R. Khandelwal, "Evaluating performance of an object detection model." https://towardsdatascience.com/evaluating-performance-of-an-object-detection-model-137a349c517b, 2021.

[57] S. Sharma, "Epoch vs batch size vs iterations." https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9, 2017.

[58] A. Startups, "Yolov8 vs. yolov5: Choosing the best object detection model." https://www.augmentedstartups.com/blog/yolov8-vs-yolov5-choosing-the-best-object-detection-model, 2023.

[59] W. Brown, "Perspective projections." http://learnwebgl.brown37.net/08_projections/projections_perspective.html, 2016.

[60] A. Singh, "Selecting the right bounding box using non-max suppression." https://www.analyticsvidhya.com/blog/2020/08/selecting-the-right-bounding-box-using-non-max-suppression-with-implementation/, 2020.

[61] PyPi, "2d bounding box conversions." https://pypi.org/project/bbox-utils/, 2023.

[62] J. W. Anders Grunnet-Jepsen, John N. Sweetser, "Tuning depth cameras for best performance," 2023.

[63] J. Courtney, *Marker-Free Human Motion Capture with Applications in Gait Analysis*. PhD thesis, 06 2005.

[64] Keivan, "What are the vertical and horizontal focal lengths?." `https://cs.stackexchange.com/questions/50198/what-are-the-vertical-and-horizontal-focal-lengths`, 2019.

[65] I. C. Mineo, "Locating the origin of the reference system." `https://support.intelrealsense.com/hc/en-us/community/posts/360047245353-Locating-the-origin-of-the-reference-system`, 2020.

[66] D. Coleman, I. Sucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study," *Journal of Software Engineering for Robotics, 5(1):3–16*, 2014.

[67] "Moveit commander documentation." `http://docs.ros.org/en/jade/api/moveit_commander/html/classmoveit_commander_1_1move__group_1_1MoveGroupCommander.html`. Accessed: 2023-05-31.

[68] S. W. Smith *et al.*, "The scientist and engineer's guide to digital signal processing," 1997.

[69] S. Butterworth *et al.*, "On the theory of filter amplifiers," 1930.

[70] J. Gao, *Nonlinear time series: semiparametric and nonparametric methods*. Springer Science & Business Media, 2007.

A Processing of coordinates of detected pipes - Python code

A.1 Initial part of code (Importing libraries and defining paths)

```
"""
@author: Adam Kuryla
"""
#Import libraries
import pyrealsense2 as rs
import argparse
from Sorting import Sorting, perspective, mean_closest_points
import os
import platform
import sys
from pathlib import Path
import numpy as np
import torch
import torch.backends.cudnn as cudnn

#Path for YOLO
FILE = Path(__file__).resolve()
ROOT = FILE.parents[0] # YOLOv5 root directory
if str(ROOT) not in sys.path:
    sys.path.append(str(ROOT)) # add ROOT to PATH
ROOT = Path(os.path.relpath(ROOT, Path.cwd())) # relative

#Import more libraries
from models.common import DetectMultiBackend
from utils.dataloaders import IMG_FORMATS, VID_FORMATS, LoadImages, LoadScreenshots, LoadStreams
from utils.general import (LOGGER, Profile, check_file, check_img_size, check_imshow,
     check_requirements, colorstr, cv2,
                        increment_path, non_max_suppression, scale_segments, print_args,
                             scale_boxes, strip_optimizer, xyxy2xywh)
from utils.plots import Annotator, colors, save_one_box
from utils.torch_utils import select_device, smart_inference_mode, time_sync

#Letterbox for resizing
def letterbox(img, new_shape=(640, 640), color=(114, 114, 114), auto=True, scaleFill=False,
     scaleup=True, stride=32):
    # Resize and pad image while meeting stride-multiple constraints
    shape = img.shape[:2] # current shape [height, width]
    if isinstance(new_shape, int):
        new_shape = (new_shape, new_shape)
    # Scale ratio (new / old)
    r = min(new_shape[0] / shape[0], new_shape[1] / shape[1])
    if not scaleup: # only scale down, do not scale up (for better test mAP)
        r = min(r, 1.0)
    # Compute padding
    ratio = r, r # width, height ratios
    new_unpad = int(round(shape[1] * r)), int(round(shape[0] * r))
    dw, dh = new_shape[1] - new_unpad[0], new_shape[0] - new_unpad[1] # wh padding
    if auto: # minimum rectangle
        dw, dh = np.mod(dw, stride), np.mod(dh, stride) # wh padding
    elif scaleFill: # stretch
        dw, dh = 0.0, 0.0
        new_unpad = (new_shape[1], new_shape[0])
        ratio = new_shape[1] / shape[1], new_shape[0] / shape[0] # width, height ratios
    dw /= 2 # divide padding into 2 sides
    dh /= 2
    if shape[::-1] != new_unpad: # resize
```

```
        img = cv2.resize(img, new_unpad, interpolation=cv2.INTER_LINEAR)
    top, bottom = int(round(dh - 0.1)), int(round(dh + 0.1))
    left, right = int(round(dw - 0.1)), int(round(dw + 0.1))
    img = cv2.copyMakeBorder(img, top, bottom, left, right, cv2.BORDER_CONSTANT, value=color) # add
         border
    return img, ratio, (dw, dh)

@smart_inference_mode()
def run(
        weights=ROOT / 'best.pt', # model path or triton URL
        source=ROOT / 'data/images', # file/dir/URL/glob/screen/0(webcam)
        data=ROOT / 'data/coco128.yaml', # dataset.yaml path
        imgsz=(640, 640), # inference size (height, width)
        conf_thres=0.85, # confidence threshold
        iou_thres=0.45, # NMS IOU threshold
        max_det=1000, # maximum detections per image
        device='', # cuda device, i.e. 0 or 0,1,2,3 or cpu
        view_img=False, # show results
        save_txt=True, # save results to *.txt
        save_conf=True, # save confidences in --save-txt labels
        save_crop=True, # save cropped prediction boxes
        nosave=False, # do not save images/videos
        classes=None, # filter by class: --class 0, or --class 0 2 3
        agnostic_nms=False, # class-agnostic NMS
        augment=False, # augmented inference
        visualize=False, # visualize features
        update=False, # update all models
        project=ROOT / 'runs/detect', # save results to project/name
        name='exp', # save results to project/name
        exist_ok=False, # existing project/name ok, do not increment
        line_thickness=0.1, # bounding box thickness (pixels)
        hide_labels=True, # hide labels
        hide_conf=True, # hide confidences
        half=False, # use FP16 half-precision inference
        dnn=False, # use OpenCV DNN for ONNX inference
        vid_stride=1, # video frame-rate stride
):
```

A.2 Activation of Real Sense camera

```
#Start Real Sense
    pipeline = rs.pipeline()
    config = rs.config()
    align = rs.align(rs.stream.color)
    config.enable_stream(rs.stream.color, 1280, 720, rs.format.bgr8, 30)
    ########################################################################
    config.enable_stream(rs.stream.depth, 1280, 720, rs.format.z16, 30)
    ########################################################################
    print("[INFO] Starting streaming...")
    profile=pipeline.start(config)
    #Get intrinsic used for perspective transformation
    intr = profile.get_stream(rs.stream.color).as_video_stream_profile().get_intrinsics()
    print("[INFO] Camera ready.")

    source = str(source)
    save_img = not nosave and not source.endswith('.txt') # save inference images
    is_file = Path(source).suffix[1:] in (IMG_FORMATS + VID_FORMATS)
    is_url = source.lower().startswith(('rtsp://', 'rtmp://', 'http://', 'https://'))
    webcam = source.isnumeric() or source.endswith('.txt') or (is_url and not is_file)
    if is_url and is_file:
        source = check_file(source) # download

    # Directories
    save_dir = increment_path(Path(project) / name, exist_ok=exist_ok) # increment run
    #(save_dir / 'labels' if save_txt else save_dir).mkdir(parents=True, exist_ok=True) # make dir
    (save_dir / 'labels').mkdir(parents=True, exist_ok=True) # make dir

```

```
    # Adjustment of camera
    device = select_device(device)
    model = DetectMultiBackend(weights, device=device, dnn=dnn, data=data, fp16=half)
    stride, names, pt = model.stride, model.names, model.pt
    imgsz = check_img_size(imgsz, s=stride) # check image size

    # Dataloader
    webcam= False
    if webcam:
        view_img = check_imshow()
        cudnn.benchmark = True # set True to speed up constant image size inference
        dataset = LoadStreams(source, img_size=imgsz, stride=stride, auto=pt)
        bs = len(dataset) # batch_size
    else:
        dataset = LoadImages(source, img_size=imgsz, stride=stride, auto=pt)
        bs = 1 # batch_size
    vid_path, vid_writer = [None] * bs, [None] * bs

    # Run inference
    model.warmup(imgsz=(1 if pt else bs, 3, *imgsz)) # warmup
    dt, seen = [0.0, 0.0, 0.0], 0
```

A.3 Prediction and detection of objects with YOLO

```
#Camera is catching frames - main part of code----------------------------------------------------------------
    cv2.waitKey(1000) #delay in case to give program some time to compile
    pipe_coord=[]
    while True:
    # for path, im, im0s, vid_cap, s in dataset:
        save_img = True
        path = os.getcwd()
        frames = pipeline.wait_for_frames()
        aligned_frames = align.process(frames)
        color_frame = aligned_frames.get_color_frame()
        depth = aligned_frames.get_depth_frame()

        if not depth: continue
        #Here model is detecting objects (prediction) and inserting boxes around
        color_image = np.asanyarray(color_frame.get_data())
        im0s = color_image
        img = letterbox(im0s)[0]
        img = img[:, :, ::-1].transpose(2, 0, 1) # BGR to RGB, to 3x416x416
        im = np.ascontiguousarray(img)
        t1 = time_sync()
        im = torch.from_numpy(im).to(device)
        im = im.half() if model.fp16 else im.float() # uint8 to fp16/32
        im /= 255 # 0 - 255 to 0.0 - 1.0
        if len(im.shape) == 3:
            im = im[None] # expand for batch dim
        t2 = time_sync()
        dt[0] += t2 - t1

        # Inference
        visualize = increment_path(save_dir / Path(path).stem, mkdir=True) if visualize else False
        pred = model(im, augment=augment, visualize=visualize)
        t3 = time_sync()
        dt[1] += t3 - t2

        # NMS
        pred = non_max_suppression(pred, conf_thres, iou_thres, classes, agnostic_nms,
            max_det=max_det)
        dt[2] += time_sync() - t3

        # Second-stage classifier (optional)
        # pred = utils.general.apply_classifier(pred, classifier_model, im, im0s)

```

```
        # Process predictions - storing detected predictions
        for i, det in enumerate(pred): # per image
            seen += 1
            if webcam: # batch_size >= 1
                p, im0, frame = path[i], im0s[i].copy(), dataset.count
                s = f'{i}: '
            else:
                p, s, im0 = path, '', im0s
                # p, im0, frame = path, im0s.copy(), getattr(dataset, 'frame', 0)

            p = Path(p) # to Path
            save_path = str(save_dir / p.name) # im.jpg
            txt_path = str(save_dir / 'labels' / p.stem) + ('' if dataset.mode == 'image' else
                f'_{frame}') # im.txt
            s += '%gx%g ' % im.shape[2:] # print string
            gn = torch.tensor(im0.shape)[[1, 0, 1, 0]] # normalization gain whwh
            imc = im0.copy() if save_crop else im0 # for save_crop
            annotator = Annotator(im0, line_width=line_thickness, example=str(names))
```

A.4 Determining of Cartesian coordinates of pipes

```
if len(det):
                # Rescale boxes from img_size to im0 size
                det[:, :4] = scale_boxes(im.shape[2:], det[:, :4], im0.shape).round()

                #Part where coordinates are detected
                centers_pipes=[] #coord of singular pipe
                pipe_coord=[] #matrix containing all defined coordinates of pipes
                depth_coord=[] #depth to tip of pipe (very often not feasible due to small edge)
                depth_iter=[] #all depths of pipes to tip
                depth_plate1=[] #depth to the plate next to detected pipe
                depth_plate2=[] #all depth to the plate
                for *xyxy, conf, cls in reversed(det):
                      c1, c2 = (int(xyxy[0]),int(xyxy[1])), (int(xyxy[2]),int(xyxy[3]))
                      center_point = round((c1[0]+c2[0])/2), round((c1[1]+c2[1])/2)
                      center_pointX = round((c1[0]+c2[0])/2)
                      center_pointY = round((c1[1]+c2[1])/2)
                      radius=4
                      depth_data = np.asanyarray(depth.get_data())
                      rows, cols = np.indices(depth_data.shape)
                      roi_mask = (np.sqrt((rows - center_pointY)**2 + (cols - center_pointX)**2) <=
                          radius)
                      depth_roi = depth_data[roi_mask]
                      depth_plate = depth_data[center_pointY,center_pointX+15]
                      depth_avg_non_zeros = list(filter(lambda x: x != 0, depth_roi.tolist()))
                      #depth_avg_non_zeros = list(filter(lambda x: x != 0, depth_avg))
                      if depth_avg_non_zeros:
                         depth_tip=min(depth_avg_non_zeros)
                      else:
                         depth_tip = 0

                      centers_pipes.append(center_point)
                      depth_coord.append(depth_tip)
                      depth_plate1.append(depth_plate)
                      circle = cv2.circle(im0,center_point,5,(0,255,0),2)
                      text_coord =
                          cv2.putText(im0,str(center_point),center_point,cv2.FONT_HERSHEY_PLAIN,2,(0,0,255))
                pipe_coord.extend(centers_pipes)
                depth_iter.extend(depth_coord)
                depth_plate2.extend(depth_plate1)
```

A.5 Initial results printed in console and default saving of YOLO model

```
# Print results in console
                for c in det[:, -1].unique():
                    n = (det[:, -1] == c).sum() # detections per class
                    s += f"{n} {names[int(c)]}{center_point}{'s' * (n > 1)}, " # add to string

                # Write results (YOLO default code)
                for *xyxy, conf, cls in reversed(det):
                    #if save_txt: # Write to file
                        xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist() #
                             normalized xywh
                        line = (cls, *xywh, conf) if save_conf else (cls, *xywh) # label format
                        with open(f'{txt_path}.txt', 'a') as f:
                            f.write(('%g ' * len(line)).rstrip() % line + '\n')

                    #if save_img or save_crop or view_img: # Add bbox to image
                        c = int(cls) # integer class
                        label = None if hide_labels else (names[c] if hide_conf else f'{names[c]}
                             {conf:.2f}')
                        annotator.box_label(xyxy, label, color=colors(c, True))
                        d1, d2 = int((int(xyxy[0])+int(xyxy[2]))/2), int((int(xyxy[1])+int(xyxy[3]))/2)
                        zDepth = depth.get_distance(int(d1),int(d2)) # by default realsense returns
                             distance in meters
                        tl = 1
                        tf = max(tl - 1, 1) # font thickness
                        cv2.putText(im0, str(round((zDepth* 100 ),2)), (d1-10, d2), 0, tl / 3, [225,
                             255, 255], thickness=tf, lineType=cv2.LINE_AA)
```

A.6 Loop break and depth filtering

```

        #Loop if broken at the point when all pipes in cooler are found on side (in this case cooler
             has 85 pipes)
        if len(pipe_coord) == 85:
           break
       #display procedure of finding in the window (Used while using on local computer)
        cv2.imshow(str(p), im0)
        cv2.waitKey(1) # 1 millisecond

        # Print time (inference-only)
        LOGGER.info(f'{s}Done. ({t3 - t2:.3f}s)')\

    #Filtering of defined depths and processing of them
    filtered_depth = list(filter(lambda x: x != 0, depth_plate2)) #some of results might be NAN so
         they are exchanged to 0
    thresholdOutlier = 400 #Criterium to exclude depth which are too big
    filtered_depth = [value for value in filtered_depth if value <= thresholdOutlier]

    thresholdMean = 20 #Criterium to exclude those values, which are too far from mean value
    valid_depth_values = [value for value in filtered_depth if abs(value - np.mean(filtered_depth))
         <= thresholdMean]
    Depth_mean = np.mean(valid_depth_values)
    #Depth_mean = sum(filtered_depth) / len(filtered_depth)
    #Exchange values of 0 to depth mean
    Depth_plate3 = [Depth_mean if value == 0 else value for value in depth_plate2]
```

A.7 Saving of data

```
#Saving part of code (All data are saved in seperate files)
```

```
    with open('pipe_centers.txt', 'w') as f: #File cointains coordites of pipes in image
     for i, center in enumerate(pipe_coord):
        depth_value = Depth_plate3[i] if i < len(Depth_plate3) else 0 # Retrieve Depth_plate3 value
             for cooresponding pipes
        if depth_value:
            f.write(f"{center[0]},{center[1]},{depth_value}\n")
        else:
            f.write(f"{center[0]},{center[1]},not found\n") # Save center coordinates with 'not
                 found' if depth value is 0


    with open('pipe_depths.txt', 'w') as f: #Files cointainning depths to tip of pipes
       for depthK in depth_iter:
           if depthK:
               f.write(f" {depthK}\n")
           else:
               f.write(f"0\n")

    with open('pipe_depths4.txt', 'w') as f: #File cointanning filter depth data to pipes (used for
         postprocessing)
       for depthK in Depth_plate3:
           if depthK:
               f.write(f" {depthK}\n")
           else:
               f.write(f"0\n")

    with open('pipe_depths3.txt', 'w') as f: #Not processed depth values (for user to see which pipes
         were not detected)
       for depthK3 in depth_plate2:
           if depthK3:
               f.write(f" {depthK3}\n")
           else:
               f.write(f"0\n")

    with open('pipe_depths2.txt', 'w') as f: #File containning parameters used for perspective
         transformation
               f.write(str(Depth_mean))
               f.write("\n")
               f.write(str(intr.ppx))
               f.write("\n")
               f.write(str(intr.ppy))
               f.write("\n")
               f.write(str(intr.fx))
               f.write("\n")
               f.write(str(intr.fy))
```

A.8 Sorting and Perspective transformation function activation

```
#Commands for sorting of pipes (Check Sorting file)
    path=r'pipe_centers.txt'
    Sorting(path)
    #Commands for perspective transformation (Check transformation function)
    path2=r'pipe_segregated.txt'
    perspective(path2,Depth_mean,intr.ppx,intr.ppy,intr.fx,intr.fy)
```

A.9 Parameters for YOLO model and end of program

```
#Parameters for YOLO model for adjustment of detection and choosing weight and classes
def parse_opt():
```

```
3       parser = argparse.ArgumentParser()
4       parser.add_argument('--weights', nargs='+', type=str, default=ROOT / 'yolov5s.pt', help='model
            path or triton URL')
5       parser.add_argument('--source', type=str, default=ROOT / 'data/images',
            help='file/dir/URL/glob/screen/0(webcam)')
6       parser.add_argument('--data', type=str, default=ROOT / 'data/coco128.yaml', help='(optional)
            dataset.yaml path')
7       parser.add_argument('--imgsz', '--img', '--img-size', nargs='+', type=int, default=[640],
            help='inference size h,w')
8       parser.add_argument('--conf-thres', type=float, default=0.85, help='confidence threshold')
9       parser.add_argument('--iou-thres', type=float, default=0.45, help='NMS IoU threshold')
10      parser.add_argument('--max-det', type=int, default=1000, help='maximum detections per image')
11      parser.add_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')
12      parser.add_argument('--view-img', action='store_true', help='show results')
13      parser.add_argument('--save-txt', action='store_true', help='save results to *.txt')
14      parser.add_argument('--save-conf', action='store_true', help='save confidences in --save-txt
            labels')
15      parser.add_argument('--save-crop', action='store_true', help='save cropped prediction boxes')
16      parser.add_argument('--nosave', action='store_true', help='do not save images/videos')
17      parser.add_argument('--classes', nargs='+', type=int, help='filter by class: --classes 0, or
            --classes 0 2 3')
18      parser.add_argument('--agnostic-nms', action='store_true', help='class-agnostic NMS')
19      parser.add_argument('--augment', action='store_true', help='augmented inference')
20      parser.add_argument('--visualize', action='store_true', help='visualize features')
21      parser.add_argument('--update', action='store_true', help='update all models')
22      parser.add_argument('--project', default=ROOT / 'runs/detect', help='save results to
            project/name')
23      parser.add_argument('--name', default='exp', help='save results to project/name')
24      parser.add_argument('--exist-ok', action='store_true', help='existing project/name ok, do not
            increment')
25      parser.add_argument('--line-thickness', default=3, type=int, help='bounding box thickness
            (pixels)')
26      parser.add_argument('--hide-labels', default=False, action='store_true', help='hide labels')
27      parser.add_argument('--hide-conf', default=False, action='store_true', help='hide confidences')
28      parser.add_argument('--half', action='store_true', help='use FP16 half-precision inference')
29      parser.add_argument('--dnn', action='store_true', help='use OpenCV DNN for ONNX inference')
30      parser.add_argument('--vid-stride', type=int, default=1, help='video frame-rate stride')
31      opt = parser.parse_args()
32      opt.imgsz *= 2 if len(opt.imgsz) == 1 else 1 # expand
33      print_args(vars(opt))
34      return opt
35
36
37  def main(opt):
38      check_requirements(exclude=('tensorboard', 'thop'))
39      run(**vars(opt))
40
41
42  if __name__ == '__main__':
43      opt = parse_opt()
44      main(opt)
```

B | Calibration of camera - Python code

```python
#!/usr/bin/env python3

import numpy as np
import cv2 as cv
import glob



################ FIND CHESSBOARD CORNERS - OBJECT POINTS AND IMAGE POINTS ##############################

chessboardSize = (9,16)
frameSize = (1440,1080)



# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)


# preparing object points
objp = np.zeros((chessboardSize[0] * chessboardSize[1], 3), np.float32)
objp[:,:2] = np.mgrid[0:chessboardSize[0],0:chessboardSize[1]].T.reshape(-1,2)

size_of_chessboard_squares_mm = 15
objp = objp * size_of_chessboard_squares_mm


# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.


# Read all images
images = glob.glob('*.jpg')


# Loop for all images
for image in images:

    img = cv.imread(image) # Load image
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) # Convert to grayscale

    # Display image
    cv.imshow('img', img)
    cv.waitKey(1000)

    # Find the chess board corners
    ret, corners = cv.findChessboardCorners(gray, chessboardSize, None)

    # If found, add object points, image points (after refining them)
    if ret == True:

        objpoints.append(objp)
        corners2 = cv.cornerSubPix(gray, corners, (11,11), (-1,-1), criteria)
        imgpoints.append(corners)

        # Draw and display the corners
        cv.drawChessboardCorners(img, chessboardSize, corners2, ret)

        # Display image with corners
        cv.imshow('img', img)
        cv.waitKey(1000)
```

```
63          # cv.imwrite('1.png', img)
64
65
66  cv.destroyAllWindows()
67
68
69  ############## CALIBRATION #######################################################
70
71  ret, cameraMatrix, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, frameSize, None,
        None)
72
73
74  # Save the cameraMatrix in a text file
75  np.savetxt('cameraMatrix.txt', cameraMatrix)
76
77  # Save the dist in a text file
78  np.savetxt('dist.txt', dist)
79
80
81  # Reprojection Error
82  mean_error = 0
83
84  for i in range(len(objpoints)):
85      imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], cameraMatrix, dist)
86      error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2)/len(imgpoints2)
87      mean_error += error
88
89  print( "total error: {}".format(mean_error/len(objpoints)) )
90
91  Re_error = mean_error / len(objpoints)
92  print(Re_error)
93
94  # Convert the number into a 1D numpy array
95  Re_error_arr = np.array([Re_error])
96
97  # Save the array into a text file
98  np.savetxt('Reprojection_Error.txt', Re_error_arr)
```

C | Sorting and perspective funtions - Python code

C.1 Sorting

```python
def Sorting(path):

    #read coordinates X,Y,Z
    df = pd.read_csv(path,delimiter=',',header=None)
    df.columns = ['X', 'Y','Z']
    Xcoord=df.iloc[:,0].values
    Ycoord=df.iloc[:,1].values
    Zcoord=df.iloc[:,2].values #Z is afterwards prescribed for X,Y coordinates
    threshold = 10

    df = df.sort_values(by=['Y'])
    differences = np.diff(df['Y'])

    #Sorting Y by values which lay in the same row
    sets = []
    current_set = [df.iloc[0]]
    for i in range(1, len(df)):
        if df.iloc[i]['Y'] - df.iloc[i-1]['Y'] <= threshold:
            current_set.append(df.iloc[i])
        else:
            sets.append(current_set)
            current_set = [df.iloc[i]]
    # Append the last set
    sets.append(current_set)
    # Sorting X (growing for odd and descending for even)
    #For each defined set of Y's, X is sorted either by decreasing or increasing manner
    for i, s in enumerate(sets):
        if i % 2 == 0: # Even set - X increase
            s.sort(key=lambda r: r['X'], reverse=True)
        else: # Odd set - X decrease
            s.sort(key=lambda r: r['X'], reverse=False)

    #Save segregated data
    with open('pipe_segregated.txt', 'w') as f:
        for i, s in enumerate(sets):
            #f.write(f'Set {i+1}:\n')
```

C.2 Perspective

```python
def perspective(path,depth,ppx,ppy,fx,fy):
    df = pd.read_csv(path,delimiter=',',header=None)
    df0=df.to_numpy()
    X_real=[]
    Y_real=[]
    Z_real=[]
    for i in range(0, len(df)):
        Xnew=df0[i,2]*(df0[i,0]-ppx)/fx
        Ynew=df0[i,2]*(df0[i,1]-ppy)/fy
        X_real.append(Xnew)
        Y_real.append(Ynew)
        Z_real.append(df0[i,2])
    print(X_real)
    #Save data
    with open('pipe_coord_real.txt', 'w') as f:
        for i in range(0,len(df)):
            f.write(f"{X_real[i]}, {Y_real[i]}, {Z_real[i]}\n")
```

D | Gazebo Digital Twin codes

D.1 Launch file

```
<launch>
    <!-- Load universal robot description format (URDF) into parameter server -->
    <param name="robot_description" command="$(find xacro)/xacro '$(find
        gazebo_digital_twin)/urdf/ur5e_mir200.xacro'" />

    <!-- Start Gazebo world -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch"/>

    <!-- push robot_description to factory and spawn robot in gazebo -->
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param robot_description -urdf
        -model robot -z 1" />

    <!-- Load joint controller configurations from YAML file to parameter server -->
    <rosparam file="$(find gazebo_digital_twin)/config/ur5e_mir200.yaml" command="load"/>

    <!-- load the controllers -->
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
        output="screen"
    args="ur5e_arm_controller mir200_base_controller"/>

        <!-- Other launch file elements -->

    <!-- node to mimic real-robot joint states in Gazebo -->
    <node name="joint_state_replicator" pkg="robot_controller" type="controller.py" output="screen"/>

    <!-- cmd_vel forwarder node -->
    <node name="cmd_vel_forwarder" type="cmd_vel_forwarder.py" pkg="robot_controller" output="screen"
        />

</launch>
```

D.2 UR5e Joint Replicator code in Python

```
#!/usr/bin/env python3

import rospy
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
from geometry_msgs.msg import Twist

rospy.init_node('robot_controller')

ur5e_joints = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint', 'wrist_1_joint',
    'wrist_2_joint', 'wrist_3_joint']

ur5e_pub = rospy.Publisher('/ur5e_arm_controller/command', JointTrajectory, queue_size=10)

def joint_state_callback(msg):
    ur5e_traj = JointTrajectory()
    ur5e_traj.header.stamp = rospy.Time.now()
    ur5e_point = JointTrajectoryPoint()

    for name, pos, vel, eff in zip(msg.name, msg.position, msg.velocity, msg.effort):
        if name in ur5e_joints:
            ur5e_traj.joint_names.append(name)
            ur5e_point.positions.append(pos)
            ur5e_point.velocities.append(vel)
            ur5e_point.effort.append(eff)
```

```
25
26      ur5e_point.time_from_start = rospy.Duration(0.1)
27      ur5e_traj.points.append(ur5e_point)
28      ur5e_pub.publish(ur5e_traj)
29
30  sub = rospy.Subscriber("/joint_states", JointState, joint_state_callback)
31
32  rospy.spin()
```

D.3 cmd_vel_forwarder code in Python

```
1   #!/usr/bin/env python3
2
3   import rospy
4   from geometry_msgs.msg import Twist
5
6   def cmd_vel_callback(data):
7       # Create a publisher for the mobile_base_manipulator/cmd_vel topic
8       pub = rospy.Publisher('mobile_base_controller/cmd_vel', Twist, queue_size=10)
9
10      # Publish the received Twist message to mobile_base_manipulator/cmd_vel
11      pub.publish(data)
12
13  def cmd_vel_forwarder():
14      # Initialize the ROS node
15      rospy.init_node('cmd_vel_forwarder', anonymous=True)
16
17      # Create a subscriber for the /cmd_vel topic
18      rospy.Subscriber("/cmd_vel", Twist, cmd_vel_callback)
19
20      # Spin the node to receive and forward messages
21      rospy.spin()
22
23  if __name__ == '__main__':
24      cmd_vel_forwarder()
```

E | Digital Twin - Processing of the signal

```python
import pandas as pd
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt, freqz
import numpy as np

# Load the CSV file
df = pd.read_csv('finalni.csv')

# Multiply the values by scalar factors
df['actual_current_0'] *= 10.1
df['actual_current_1'] *= 10.1
df['actual_current_2'] *= 10.1
df['actual_current_3'] *= 8.1
df['actual_current_4'] *= 8.1
df['actual_current_5'] *= 8.1

# Define the row range you want to process
start_row = 6000 # Start row index
end_row = 8000 # End row index

# Extract the desired signals
signals = {
    'actual_current_0': df['actual_current_0'][start_row:end_row+1].values,
    'actual_current_1': df['actual_current_1'][start_row:end_row+1].values,
    'actual_current_2': df['actual_current_2'][start_row:end_row+1].values,
    'actual_current_3': df['actual_current_3'][start_row:end_row+1].values,
    'actual_current_4': df['actual_current_4'][start_row:end_row+1].values,
    'actual_current_5': df['actual_current_5'][start_row:end_row+1].values
}

# Define the filter parameters
cutoff_frequency = 1 # Adjust cutoff frequency as needed
sampling_frequency = 60 # Adjust sampling frequency as needed
nyquist_frequency = 0.5 * sampling_frequency
normal_cutoff = cutoff_frequency / nyquist_frequency

# Design the low-pass filter
b, a = butter(2, normal_cutoff, btype='low', analog=False)

# Create time axis values in seconds
time = np.arange(start_row, end_row + 1) / sampling_frequency

# Figure 1: Joint Torques before and after filtering
plt.figure(figsize=(12, 8))

for i, (column, signal) in enumerate(signals.items()):
    # Apply low-pass filter
    filtered_signal = filtfilt(b, a, signal)

    plt.subplot(2, 3, i+1)
    plt.plot(time, signal, label='Original Signal')
    plt.plot(time, filtered_signal, label='Filtered Signal')
    plt.title(f'Joint {i+1} Torque')
    plt.xlabel('Time [s]')
    plt.ylabel('Torque [Nm]')
    plt.legend()
    plt.grid(True)

plt.tight_layout()

# Figure 2: One-Sided Amplitude Spectrum
plt.figure(figsize=(8, 6))
```

```
63
64  for i, (column, signal) in enumerate(signals.items()):
65      # Apply low-pass filter
66      filtered_signal = filtfilt(b, a, signal)
67
68      freq = np.fft.fftfreq(len(filtered_signal), d=1 / sampling_frequency)
69      mask = freq >= 0
70      amplitude_spectrum = np.abs(np.fft.fft(filtered_signal))
71
72      plt.plot(freq[mask], amplitude_spectrum[mask], label=f'Joint {i+1} Amplitude Spectrum')
73
74  plt.title('One-Sided Amplitude Spectrum')
75  plt.xlabel('Frequency [Hz]')
76  plt.ylabel('Amplitude')
77  plt.legend()
78  plt.grid(True)
79  plt.xlim([0, 1.7])
80  plt.ylim([0, 10000])
81
82  # Figure 3: Frequency Response
83  frequencies, response = freqz(b, a, worN=8000, fs=sampling_frequency)
84
85  plt.figure(figsize=(8, 6))
86  plt.plot(frequencies, abs(response))
87  plt.title('Frequency Response')
88  plt.xlabel('Frequency [Hz]')
89  plt.ylabel('Magnitude')
90  plt.grid(True)
91
92  # Display the plots
93  plt.show()
```

F | Drill motor control - Code

```python
import serial
import time
import subprocess

def run_arduino_cli(sketch_path, port, fqbn):
    """
    Compiles and uploads an Arduino sketch using the Arduino CLI.

    Parameters:
    sketch_path: str - The path to the .ino file.
    port: str - The port the Arduino is connected to, e.g., "COM3" on Windows or "/dev/ttyACM0" on Linux.
    fqbn: str - The fully qualified board name, e.g., "arduino:avr:uno" for an Arduino Uno.

    Returns: None
    """
    compile_command = ["arduino-cli", "compile", "--fqbn", fqbn, sketch_path]
    upload_command = ["arduino-cli", "upload", "-p", port, "--fqbn", fqbn, sketch_path]

    # Compile the sketch
    result = subprocess.run(compile_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    if result.returncode != 0:
        print(f"Compilation failed: {result.stderr.decode('utf-8')}")
        return


    print(f"Compilation successful: {result.stdout.decode('utf-8')}")

    # Upload the sketch
    result = subprocess.run(upload_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    if result.returncode != 0:
        print(f"Upload failed: {result.stderr.decode('utf-8')}")
        return

    print(f"Upload successful: {result.stdout.decode('utf-8')}")

# Example usage
run_arduino_cli("C:/Users/anton/OneDrive - Aarhus universitet/Master Thesis/Thesis/Antonio/Codes/Drill control/versions/1drill_control/arduino_drill_control/arduino_drill_control.ino", "COM7", "arduino:avr:uno")
arduino = serial.Serial('COM7', 9600) # Create Serial port object called arduino
time.sleep(2) # wait for 2 secounds for the communication to get established

def set_motor_rpm(rpm):
    command = str(rpm) + '\n' # convert the rpm to string and add a newline
    arduino.write(command.encode()) # send the command to the arduino

# Example usage:
set_motor_rpm(15000) # starts the motor at 1000 RPM
input("Press Enter...")
# time.sleep(10) # let the motor run for 10 seconds
set_motor_rpm(0) # stop the motor
```

```arduino
const int PUL = 3; //define Pulse pin
const int DIR = 2; //define Direction pin
int rpm = 0;
long pulseDelay = 0;
String inputString = "";      // a string to hold incoming data
bool stringComplete = false; // whether the string is complete

void setup() {
  pinMode(PUL, OUTPUT);
  pinMode(DIR, OUTPUT);
  Serial.begin(9600); // Starts the serial communication
```

```
12    inputString.reserve(10); // reserve 10 bytes for the inputString
13  }
14
15  void loop() {
16    if (stringComplete) {
17      rpm = inputString.toInt();
18      inputString = "";
19      stringComplete = false;
20      if (rpm != 0) {
21        pulseDelay = (long)((60.0 * 1000000.0) / (800.0 * abs(rpm))); // Calculate delay based on new
              RPM
22      }
23    }
24
25    if (rpm != 0) {
26      digitalWrite(DIR, rpm > 0 ? HIGH : LOW); // Set the rotation direction
27      digitalWrite(PUL,HIGH);
28      delayMicroseconds(pulseDelay/2);
29      digitalWrite(PUL,LOW);
30      delayMicroseconds(pulseDelay/2);
31    }
32  }
33
34  void serialEvent() {
35    while (Serial.available()) {
36      char inChar = (char)Serial.read();
37      inputString += inChar;
38      if (inChar == '\n') {
39        stringComplete = true;
40      }
41    }
42  }
```

G Optimization of UR5e joint configuration - MATLAB code

```
clear
clc
close all

% import Robotics System Toolbox
import robotics.*

% Define the DH parameters for UR5e
L(1) = Revolute('d', 0.1625, 'a', 0, 'alpha', pi/2, 'm', 3.761, 'r', [0,-0.02561,0.00193], 'I', [0.010267495893,0.010267495893,0.00666]);
L(2) = Revolute('d', 0, 'a', -0.425, 'alpha', 0, 'm', 8.058, 'r', [0.2125,0,0.11336], 'I', [0.22689067591,0.22689067591,0.0151074]);
L(3) = Revolute('d', 0, 'a', -0.3922, 'alpha', 0, 'm', 2.846, 'r', [0.15,0,0.0265], 'I', [0.049443313556,0.049443313556,0.004095]);
L(4) = Revolute('d', 0.1333, 'a', 0, 'alpha', pi/2, 'm', 1.37, 'r', [0,-0.0018,0.01634], 'I', [0.111172755531,0.111172755531,0.21942]);
L(5) = Revolute('d', 0.0997, 'a', 0, 'alpha', -pi/2, 'm', 1.3, 'r', [0,0.0018,0.01634], 'I', [0.111172755531,0.111172755531,0.21942]);
L(6) = Revolute('d', 0.0996, 'a', 0, 'alpha', 0, 'm', 0.365, 'r', [0,0,-0.001159], 'I', [0.0171364731454,0.0171364731454,0.033822]);

% Create a robot model
UR5e = SerialLink(L, 'name', 'UR5e');

% Drill as a new link
L_drill = Link('d', 0, 'a', 0, 'alpha', 0, 'm', 2, 'r', [-0.08869, 0.00282, -0.05050], 'I', [ 0.00865, 0.02701, 0.01907]);

% Drill to the robot model
L_new = [L, L_drill];
UR5e_drill = SerialLink(L_new, 'name', 'UR5e_with_drill');

% Gravity vector
UR5e.gravity = [0 0 -9.81];

% Ddesired end-effector pose
T_desired = transl(0.0041, 0.45458, 0.01869) * rpy2tr( 4.794, 0.063, 0.135, 'XYZ');

% Set the optimization options
options = optimoptions('fmincon', 'Display', 'off');

% Tthe weights for the pose error, the torque difference, and the last joint angle
w_pose = 1000;
w_torque = 0.01;
w_joint6 = 1000;

% Initial configuration for consideration
q_initial = [0 -pi/2  0   -pi/2  0    0]'

% Torques in the joints at the initial configuration
tau_g_initial = UR5e_drill.gravload([q_initial; 0]');
torques_initial = tau_g_initial;

% Position and orientation error cost function for optimization
pose_cost_function = @(q) norm(UR5e_drill.fkine([q;0]).t() - T_desired(1:3, 4)) + ...
                          norm(tr2rpy(UR5e_drill.fkine([q;0]).R()) - tr2rpy(T_desired(1:3,1:3)));

% Estimated drilling forces and torques
Fx=7.27; Fy=7.27; Fz=42.27;
Tx=0.05; Ty=0.05; Tz=0.22;

% Offset of the drill tip from the end-effector's origin
```

```
drill_offset = [-0.0976; 0; 0.129];

% Cost function
cost_function = @(q) w_pose*pose_cost_function(q) + w_torque*norm(torques_initial -
    UR5e_drill.gravload([q;0]') - UR5e_drill.jacob0([q; 0], 'rpy',
    drill_offset)'*drilling_forces_torques) + w_joint6*abs(q(6));

% Number of random starting points
num_starts = 100;

% Initialize the best cost and solution
best_cost = Inf;
best_solution = [];
% Initialize the array to store the cost values
cost_values = [];

% Perform optimization from multiple starting points
for i = 1:num_starts
    % Generate a random initial configuration within the joint limits
    q_initial = (UR5e.qlim(:,2) - UR5e.qlim(:,1)).*rand(6,1) + UR5e.qlim(:,1);
    % First joint is always negative
    q_initial(1) = -abs(q_initial(1));

    % Perform optimization to find the optimal joint configuration
    UR5e.qlim(1,2) = min(0, UR5e.qlim(1,2));
    [q_solution, cost] = fmincon(cost_function, q_initial, [], [], [], [], UR5e.qlim(:,1),
        UR5e.qlim(:,2), [], options);

    % Store the cost value
    cost_values = [cost_values; cost];

    % If the cost is lower than the best cost, update the best cost and solution
    if cost < best_cost
        best_cost = cost;
        best_solution = q_solution;
    end
end

% Use the best solution as the optimal configuration
q_solution = best_solution';

% Compute the torques due to gravity at the optimal configuration
tau_g = UR5e_drill.gravload([q_solution, 0]);

% Compute the Jacobian at the offset position for optimal configuration
J_offset = UR5e_drill.jacob0([q_solution, 0], 'rpy', drill_offset);

% Compute the torques due to drilling at the optimal configuration
tau_drill = J_offset' * drilling_forces_torques;

% Compute the total torques in the joints at the optimal configuration
torques_optimal = tau_g + tau_drill';

% Display the initial and optimal joint configurations and torques
fprintf('\nInitial configuration: %f %f %f %f %f %f\n', q_initial)
fprintf('Initial torques: %f %f %f %f %f %f %f\n', torques_initial)
fprintf('\nOptimal configuration: %f %f %f %f %f %f\n', q_solution)
fprintf('Optimal torques: %f %f %f %f %f %f %f\n', torques_optimal)

UR5e_optimal = SerialLink(L, 'name', 'UR5e');

% Optimal configuration
figure(1);
UR5e_optimal.plot(q_solution);
title('UR5e at Optimal Configuration');
ee_pose_optimal = UR5e_optimal.fkine(q_solution);
ee_t_optimal = transl(ee_pose_optimal); % extract translation part (end effector position)
text(-1, 1, 1, sprintf('End Effector: [%.4f, %.4f, %.4f]', ee_t_optimal));
```

H | Alignment of end-effector with checkerboard - Python code

```python
#!/usr/bin/env python3

import cv2 as cv
import numpy as np
import moveit_commander
import geometry_msgs.msg
import sys
import copy
import math
import rospy
import moveit_msgs.msg
from geometry_msgs.msg import Pose, Quaternion
import tf
from moveit_msgs.msg import DisplayTrajectory, RobotState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
import pyrealsense2 as rs
from scipy.spatial.transform import Rotation
import time

# Initialize MoveIt commander and UR5e group
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('chamfering')

robot = moveit_commander.RobotCommander()
group = moveit_commander.MoveGroupCommander("manipulator")

# Set the reference frame for the robot
group.set_pose_reference_frame('base')

# Set the maximum velocity and acceleration scaling factors
group.set_max_velocity_scaling_factor(0.1)
group.set_max_acceleration_scaling_factor(0.1)

# Define the DH parameters for the UR5e robot
a = [0.0, -0.42500, -0.39225, 0.0, 0.0, 0.0]
# d = [0.089159, 0.0, 0.0, 0.10915, 0.09465, 0.0823]
d = [0.1625, 0.0, 0.0, 0.1333, 0.0997, 0.0996]

alpha = [math.pi/2, 0.0, 0.0, math.pi/2, -math.pi/2, 0.0]

def deg_to_rad(deg):
    return deg * math.pi / 180

# Set the joint goal
joint_goal = group.get_current_joint_values()
joint_goal[0] = deg_to_rad(-114.7)
joint_goal[1] = deg_to_rad(-19.20)
joint_goal[2] = deg_to_rad(119.10)
joint_goal[3] = deg_to_rad(78.1)
joint_goal[4] = deg_to_rad(-57.80)
joint_goal[5] = 0

# Plan and execute the trajectory
group.go(joint_goal, wait=True)

# Joint angles in degrees
theta_deg = [-112.40, -18.70, 118.10, -284.70, -68.30, 0.00]
theta = np.deg2rad(theta_deg)

# Compute the transformation matrix for the end effector
T_06 = np.identity(4)
for i in range(6):
```

```
63      c_theta = math.cos(theta[i])
64      s_theta = math.sin(theta[i])
65      c_alpha = math.cos(alpha[i])
66      s_alpha = math.sin(alpha[i])
67      T_i = np.array([[c_theta, -s_theta*c_alpha, s_theta*s_alpha, a[i]*c_theta],
68                      [s_theta, c_theta*c_alpha, -c_theta*s_alpha, a[i]*s_theta],
69                      [0.0, s_alpha, c_alpha, d[i]],
70                      [0.0, 0.0, 0.0, 1.0]])
71      T_06 = T_06 @ T_i
72
73  # Extract the position of the end effector
74  ee_pos = T_06[:3, 3]
75  T = T_06
76
77  # Extract position vector from transformation matrix
78  pos = T[:3,3]
79
80  def transform_to_pos_quat(T):
81
82      # Extract rotation matrix from transformation matrix
83      R = T[:3,:3]
84
85      # Calculate quaternion from rotation matrix
86      tr = np.trace(R)
87      if tr > 0:
88          S = math.sqrt(tr+1.0) * 2 # S=4*qw
89          qw = 0.25 * S
90          qx = (R[2,1] - R[1,2]) / S
91          qy = (R[0,2] - R[2,0]) / S
92          qz = (R[1,0] - R[0,1]) / S
93      elif (R[0,0] > R[1,1]) and (R[0,0] > R[2,2]):
94          S = math.sqrt(1.0 + R[0,0] - R[1,1] - R[2,2]) * 2 # S=4*qx
95          qw = (R[2,1] - R[1,2]) / S
96          qx = 0.25 * S
97          qy = (R[0,1] + R[1,0]) / S
98          qz = (R[0,2] + R[2,0]) / S
99      elif R[1,1] > R[2,2]:
100         S = math.sqrt(1.0 + R[1,1] - R[0,0] - R[2,2]) * 2 # S=4*qy
101         qw = (R[0,2] - R[2,0]) / S
102         qx = (R[0,1] + R[1,0]) / S
103         qy = 0.25 * S
104         qz = (R[1,2] + R[2,1]) / S
105     else:
106         S = math.sqrt(1.0 + R[2,2] - R[0,0] - R[1,1]) * 2 # S=4*qz
107         qw = (R[1,0] - R[0,1]) / S
108         qx = (R[0,2] + R[2,0]) / S
109         qy = (R[1,2] + R[2,1]) / S
110         qz = 0.25 * S
111
112     # Return position and quaternion as a tuple
113     return np.array([qx, qy, qz, qw])
114
115 quat = transform_to_pos_quat(T)
116
117 ################ FIND CHESSBOARD CORNERS - OBJECT POINTS AND IMAGE POINTS
        ##############################
118 # Define desired resolution
119 width = 1280
120 height = 720
121
122 # Create a pipeline object to manage the streaming and processing of frames
123 pipeline = rs.pipeline()
124
125 # Configure the pipeline with a device and a stream
126 config = rs.config()
127 config.enable_stream(rs.stream.color, width, height, rs.format.bgr8, 15)
128
129 # Start the pipeline
130 pipeline.start(config)
131
```

```
# Wait for camera to adjust
time.sleep(4)

# Now capture the frame
# Wait for a coherent pair of frames: depth and color
frames = pipeline.wait_for_frames()

# Get the color frame
color_frame = frames.get_color_frame()

# Convert the color frame to a numpy array
color_image = np.asanyarray(color_frame.get_data())

# Save the image to disk
cv.imwrite("/home/toni/ws_moveit3/src/final_codes/image_for_calibration.jpg",
    color_image,[cv.IMWRITE_JPEG_QUALITY, 100])

# Stop the pipeline
pipeline.stop()

chessboardSize = (9,16)
frameSize = (1280,720)

# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((chessboardSize[0] * chessboardSize[1], 3), np.float32)
objp[:,:2] = np.mgrid[0:chessboardSize[0],0:chessboardSize[1]].T.reshape(-1,2)

size_of_chessboard_squares_mm = 15
objp = objp * size_of_chessboard_squares_mm

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

# Load the cameraMatrix from the text file
cameraMatrix = np.loadtxt('/home/toni/ws_moveit3/src/final_codes/Calibration/avg_cam_matrix.txt')

# Load the dist from the text file
dist = np.loadtxt('/home/toni/ws_moveit3/src/final_codes/Calibration/avg_dist_coeffs.txt')

# img = cv.imread('17_Color.png')
img = cv.imread('/home/toni/ws_moveit3/src/final_codes/image_for_calibration.jpg')

################ OBTAINING TRANSFORMATION MATRIX - CAMERA TO CHECKERBOARD
    ##############################

# Upload new image for processing
img = cv.imread('/home/toni/ws_moveit3/src/final_codes/image_for_calibration.jpg')

# find chessboard corners
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
ret, corners = cv.findChessboardCorners(gray, chessboardSize, None)

# Estimate checkerboard pose
retval, rvec, tvec = cv.solvePnP(objp, corners, cameraMatrix, dist)

# Draw the coordinate frame on the image
axisLength = 50 # length of each axis in meters
axisPoints = np.float32([[0,0,0], [axisLength,0,0], [0,axisLength,0], [0,0,axisLength]]).reshape(-1,
    3)
axisPointsImg, _ = cv.projectPoints(axisPoints, rvec, tvec, cameraMatrix, dist)
img_frames = cv.drawFrameAxes(img, cameraMatrix, dist, rvec, tvec, axisLength)

# Display the image with the coordinate frame
cv.imshow('Image with coordinate frame', img_frames)
cv.waitKey(0)
cv.destroyAllWindows()
```

```
199
200 # Convert rotation vector to rotation matrix
201 R, _ = cv.Rodrigues(rvec)
202
203 # Transformation matrix from the checkerboard to the camera frame
204 T = np.eye(4)
205 T[:3, :3] = R
206 T[:3, 3] = tvec.squeeze() / 1000.0
207
208 np.savetxt('/home/toni/ws_moveit3/src/final_codes/checkerboard_to_camera_transform.txt', T)
209
210 ############################## Alignment of end effector with checkerboard
        ########################################
211
212 # Load the transformation matrix from checkerboard to camera
213 checkerboard_to_camera_transform =
        np.loadtxt('/home/toni/ws_moveit3/src/final_codes/checkerboard_to_camera_transform.txt')
214
215 end_effector_to_camera_transform = np.array([ [ 0.99961536, -0.02511644, -0.01175969, -0.02912 ],
216                         [ 0.02572843, 0.998146,  0.05515989, -0.07788 ],
217                         [ 0.01035247, -0.05544124, 0.99840828, -0.00418 ],
218                         [ 0,          0,          0,          1        ] ])
219
220  # Get the transformation matrix from base to the wrist_3_link frame using the tf package
221 listener = tf.TransformListener()
222 listener.waitForTransform("base", "wrist_3_link", rospy.Time(), rospy.Duration(4.0))
223 (trans, rot) = listener.lookupTransform("base", "wrist_3_link", rospy.Time(0))
224 base_frame_to_end_effector_transform = np.dot(tf.transformations.translation_matrix(trans),
        tf.transformations.quaternion_matrix(rot))
225
226 # Compute the transformation from base to checkerboard
227 base_to_end_effector = np.dot(base_frame_to_end_effector_transform, end_effector_to_camera_transform)
228 base_to_checkerboard = np.dot(base_to_end_effector, checkerboard_to_camera_transform)
229
230 # Calculate new quaternion from rotation matrix
231 qx_new, qy_new, qz_new, qw_new = transform_to_pos_quat(base_to_checkerboard)
232
233 # Return position and quaternion as a tuple
234 quat_new = np.array([qx_new, qy_new, qz_new, qw_new])
235
236 # Set the target pose for the end effector
237 target_pose = Pose()
238 target_pose.position.x = pos[0]
239 target_pose.position.y = pos[1]
240 target_pose.position.z = pos[2]
241 target_pose.orientation = Quaternion(quat_new[0], quat_new[1], quat_new[2], quat_new[3])
242 group.set_pose_target(target_pose)
243
244 ################################# Obtaining the difference in rotation of X and Y around Z axis
        ############################################
245
246 def extract_euler_angles(rotation_matrix):
247     # Convert rotation matrix to Euler angles (roll, pitch, yaw)
248     r = Rotation.from_matrix(rotation_matrix)
249     euler_angles = r.as_euler('xyz', degrees=True)
250     return euler_angles
251
252 # Extract rotation matrices
253 R_T = T[:3, :3]
254 R_base_to_checkerboard = base_to_checkerboard[:3, :3]
255
256 # Calculate relative rotation matrix
257 R_relative = R_T.T @ R_base_to_checkerboard
258
259 # Extract Euler angles (roll, pitch, yaw)
260 relative_euler_angles = extract_euler_angles(R_relative)
261 # Convert angles to radians
262 roll, pitch, yaw = np.radians(relative_euler_angles)
263
264 # Print the relative rotations
```

```
print("Relative rotations (roll, pitch, yaw in radians):")
print("Roll:", roll)
print("Pitch:", pitch)
print("Yaw:", yaw)

################################### Rotating end effector to get camera in good position ###########################################################
# Define the angle of rotation in radians
theta = -pitch

# Define the rotation matrix around the Z-axis
R_rotated_ee = tf.transformations.rotation_matrix(theta, (0, 0, 1))
print("R_rotated", R_rotated_ee)

# Define the transformation matrix from the rotated end-effector to the original end-effector
T_rotated = np.identity(4)
# print("T_rotated", T_rotated)

T_rotated[:4, :4] = R_rotated_ee
# print("T_rotated", T_rotated)

# Define the transformation matrix from the base to the rotated end-effector
end_effector_rotation_transform = np.dot(base_to_checkerboard , T_rotated)

# Calculate new quaternion from rotation matrix
qx_new2, qy_new2, qz_new2, qw_new2 = transform_to_pos_quat(end_effector_rotation_transform)

# Return position and quaternion as a tuple
quat_new2 = np.array([qx_new2, qy_new2, qz_new2, qw_new2])

# Set the target pose for the end effector
target_pose = Pose()
target_pose.position.x = pos[0]
target_pose.position.y = pos[1]
target_pose.position.z = pos[2]

target_pose.orientation = Quaternion(quat_new2[0], quat_new2[1], quat_new2[2], quat_new2[3])
group.set_pose_target(target_pose)

# Plan and execute the motion
plan = group.go(wait=True)
group.stop()
group.clear_pose_targets()

########################### Set last joint to 0 degrees ###################################

# Set the joint goal
joint_goal = group.get_current_joint_values()
joint_goal[5] = 0 # Set wrist 3 angle to 0 degrees

# Plan and execute the trajectory
group.go(joint_goal, wait=True)

# Shutdown MoveIt commander
moveit_commander.roscpp_shutdown()
```

I | Chamfering of pipes - Python code

```python
#!/usr/bin/env python3

import numpy as np
import moveit_commander
import geometry_msgs.msg
import sys
import copy
import math
import rospy
import moveit_msgs.msg
from geometry_msgs.msg import Pose, Quaternion
from geometry_msgs.msg import PoseStamped
import tf
from moveit_msgs.msg import DisplayTrajectory, RobotState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
import subprocess
import time

# Initialize MoveIt commander and UR5e group
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('chamfering')

robot = moveit_commander.RobotCommander()
group = moveit_commander.MoveGroupCommander("manipulator")
group.allow_replanning(True)

# Set the reference frame for the robot
group.set_pose_reference_frame('base')

# Set the maximum velocity and acceleration scaling factors
group.set_max_velocity_scaling_factor(0.03)
group.set_max_acceleration_scaling_factor(0.01)

def get_current_pose_in_frame(group, frame_id):
    listener = tf.TransformListener()
    listener.waitForTransform(frame_id, group.get_planning_frame(), rospy.Time(0),
        rospy.Duration(4.0))

    current_pose = group.get_current_pose().pose
    pose_stamped = PoseStamped()
    pose_stamped.header.frame_id = group.get_planning_frame()
    pose_stamped.pose = current_pose

    pose_in_frame = listener.transformPose(frame_id, pose_stamped).pose

    return pose_in_frame

def pose_to_homogeneous_transform(pose):
    trans = [pose.position.x, pose.position.y, pose.position.z]
    rot = [pose.orientation.x, pose.orientation.y, pose.orientation.z, pose.orientation.w]
    return np.dot(tf.transformations.translation_matrix(trans),
        tf.transformations.quaternion_matrix(rot))

def homogeneous_transform_to_pose(T):
    trans = tf.transformations.translation_from_matrix(T)
    rot = tf.transformations.quaternion_from_matrix(T)
    pose = geometry_msgs.msg.Pose()
    pose.position.x = trans[0]
    pose.position.y = trans[1]
    pose.position.z = trans[2]
    pose.orientation.x = rot[0]
    pose.orientation.y = rot[1]
    pose.orientation.z = rot[2]
```

```
62      pose.orientation.w = rot[3]
63      return pose
64
65  # Read pipe coordinates from text file
66  pipe_coords_mm = np.loadtxt('/home/toni/ws_moveit3/src/final_codes/pipe_coord_real.txt',
        delimiter=',')
67  np.savetxt('/home/toni/ws_moveit3/src/final_codes/pipe_coords_mm.txt', pipe_coords_mm)
68
69  subprocess.Popen(['python3','coordinates_plot.py'])
70  input("Press Enter to continue...")
71
72  # Convert coordinates from mm to m
73  pipe_coords = pipe_coords_mm / 1000
74
75  depth_value = pipe_coords[:, 2]
76  mean_value_depth = np.mean(depth_value)
77
78  t_camera_to_ee = np.array([ [ 0.99961536, -0.02511644, -0.01175969, -0.02912 ],
79                              [ 0.02572843, 0.998146,  0.05515989, -0.07788 ],
80                              [ 0.01035247, -0.05544124, 0.99840828, -0.00418 ],
81                              [ 0,          0,          0,          1        ] ])
82
83  t_ee_to_drill = np.array([ [ 1, 0, 0, -0.0950 ],
84                             [ 0, 1, 0, 0.00    ],
85                             [ 0, 0, 1, 0.129 ],
86                             [ 0, 0, 0, 1     ]])
87
88  np.savetxt('/home/toni/ws_moveit3/src/final_codes/pipe_coords.txt', pipe_coords)
89
90  # Get the transformation matrix from the end-effector to the base_link frame using the tf package
91  listener = tf.TransformListener()
92  listener.waitForTransform("base", "wrist_3_link", rospy.Time(), rospy.Duration(4.0))
93  (trans, rot) = listener.lookupTransform("base", "wrist_3_link", rospy.Time(0))
94  t_ee_to_base = np.dot(tf.transformations.translation_matrix(trans),
        tf.transformations.quaternion_matrix(rot))
95  print(t_ee_to_base)
96
97  # Create an empty array to store the pipe positions in the base_link frame
98  p_pipe_in_base_array = np.empty((0, 3), float)
99
100 # Initialize an empty list to store the transformed Z values
101 transformed_z_values = []
102
103 # Loop through each pipe center and move the end-effector
104 for i in range(pipe_coords.shape[0]):
105
106     # Get the current pipe center coordinates
107     x, y, z = pipe_coords[i]
108
109     # Apply camera to end-effector transformation to get the pipe center position in the end-effector
            frame
110     p_pipe_in_ee = np.dot(t_camera_to_ee, np.array([x, y, z, 1]))
111
112     # Get the transformed Z value
113     transformed_z = p_pipe_in_ee[2]
114
115     # Append the transformed Z value to the list
116     transformed_z_values.append(transformed_z)
117
118 # Calculate the mean value of the transformed Z values
119 mean_value_z = np.mean(transformed_z_values)
120 print("mean_value_z", mean_value_z)
121
122 # Loop through each pipe center and move the end-effector
123 for i in range(pipe_coords.shape[0]):
124
125     # Current pipe center coordinates
126     x, y, z = pipe_coords[i]
127
128     # Camera to end-effector transformation to get the pipe center position in the end-effector frame
```

```
129     p_pipe_in_ee = np.dot(t_camera_to_ee, np.array([x, y, z, 1]))
130 
131     # p_pipe_in_ee[2] = mean_value_z+0.012
132 
133     # Convertion of the pipe position in the end-effector frame to a homogeneous transformation matrix
134     T_pipe_in_ee = tf.transformations.translation_matrix(p_pipe_in_ee[:-1]) # Remove last element
            because translation_matrix expects a 3D vector
135 
136     # Subtraction of the translation components of t_ee_to_drill from T_pipe_in_ee
137     T_pipe_in_ee[:-1, 3] -= t_ee_to_drill[:-1, 3]
138 
139     # Pipe position in the base_link frame
140     T_pipe_in_base = np.dot(t_ee_to_base, T_pipe_in_ee)
141 
142     p_pipe_in_base = tf.transformations.translation_from_matrix(T_pipe_in_base)
143 
144     # Append the current pipe position in the base_link frame to the array
145     p_pipe_in_base_array = np.vstack((p_pipe_in_base_array, p_pipe_in_base))
146 
147 # Drill start
148 subprocess.Popen(['python3','drill_control.py']).wait()
149 time.sleep(1)
150 
151 np.savetxt('/home/toni/ws_moveit3/src/final_codes/p_pipe_in_base_array.txt', p_pipe_in_base_array)
152 
153 # Getting current position
154 starting_pose = geometry_msgs.msg.Pose()
155 starting_pose = get_current_pose_in_frame(group, 'base')
156 
157 # Loop through each pipe center and move the end-effector
158 for i in range(p_pipe_in_base_array.shape[0]):
159     print(i)
160 
161     # # Create the target pose
162     pose_target = geometry_msgs.msg.Pose()
163     pose_target.position.x = p_pipe_in_base_array[i][0]# + 0.0966 # to position tip of the tool in
            desired point
164     pose_target.position.y = p_pipe_in_base_array[i][1] #- 0.1197-0.02 # to position tip of the tool
            in desired point
165     pose_target.position.z = p_pipe_in_base_array[i][2]
166     pose_target.orientation = starting_pose.orientation
167 
168     # Set the new pose as the target pose for the end-effector
169     group.set_pose_target(pose_target)
170 
171     # Plan and execute the motion to the new pose
172     plan = group.go(wait=True)
173     group.stop()
174     group.clear_pose_targets()
175 
176 ################################## Drilling ###################################
177 
178     # Get the current pose of the end effector in the base frame
179     current_pose_base = get_current_pose_in_frame(group, 'base')
180 
181     # Convertin in the current pose to a homogeneous transformation matrix
182     T_base_to_current = pose_to_homogeneous_transform(current_pose_base)
183 
184     # Inverse of the transformation matrix from the base to the end effector
185     T_ee_to_base = np.linalg.inv(t_ee_to_base)
186 
187     # Converting the current pose from the base frame to the end effector's frame
188     T_ee_to_current = np.dot(T_ee_to_base, T_base_to_current)
189 
190     # In the end effector frame, moving in Z
191     T_ee_to_current[2, 3] += 0.0305
192 
193     # Converting the new pose from the end-effector frame back to the base frame
194     T_base_to_new = np.dot(t_ee_to_base, T_ee_to_current)
195 
```

```
    # Convertin the new pose from a homogeneous transformation matrix to a Pose message
    new_pose_base = homogeneous_transform_to_pose(T_base_to_new)

    # Slow down the robot for drilling
    group.set_max_velocity_scaling_factor(0.01)
    group.set_max_acceleration_scaling_factor(0.01)

    group.set_pose_target(new_pose_base)

    # Plan and execute the motion
    plan = group.go(wait=True)
    group.stop()
    group.clear_pose_targets()

    time.sleep(1.5)

################################## Returning from drilling ##################################

    # Move the end effector back by the same amount

    # In the end effector frame, retracting in Z
    T_ee_to_current[2, 3] -= 0.0295

    # Convertin the new pose from the end effector frame back to the base frame
    T_base_to_new = np.dot(t_ee_to_base, T_ee_to_current)

    # Convert the new pose from a homogeneous transformation matrix to a Pose message
    new_pose_base = homogeneous_transform_to_pose(T_base_to_new)

    # Speed up the robot for moving back
    group.set_max_velocity_scaling_factor(0.02)
    group.set_max_acceleration_scaling_factor(0.01)

    group.set_pose_target(new_pose_base)

    # Plan and execute the motion
    plan = group.go(wait=True)
    group.stop()
    group.clear_pose_targets()

# Drill end
subprocess.Popen(['python3','drill_control_2.py']).wait()
time.sleep(1)

################################## Returning from drilling ##################################

# In the end effector frame, retract Z for safety
T_ee_to_current[2, 3] -= 0.17

# Converting the new pose from the end effector's frame back to the base frame
T_base_to_new = np.dot(t_ee_to_base, T_ee_to_current)

# Converting the new pose from a homogeneous transformation matrix to a Pose message
new_pose_base = homogeneous_transform_to_pose(T_base_to_new)

# Speed up the robot for moving back
group.set_max_velocity_scaling_factor(0.07)
group.set_max_acceleration_scaling_factor(0.01)

group.set_pose_target(new_pose_base)

# Plan and execute the motion
plan = group.go(wait=True)
group.stop()
group.clear_pose_targets()

# Go to home position
joint_goal = group.get_current_joint_values()
joint_goal[0] = 0
joint_goal[1] = -math.pi/2
```

```
joint_goal[2] = 0
joint_goal[3] = -math.pi/2
joint_goal[4] = 0
joint_goal[5] = 0

# Plan and execute the trajectory
group.go(joint_goal, wait=True)

# Shutdown MoveIt commander
moveit_commander.roscpp_shutdown()
```

J | MiR200 control - Python code

```python
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import tf
import math

class MiR200Controller:

    def __init__(self):
        # Initialize the node
        rospy.init_node('mir200_controller')

        # Create a publisher for cmd_vel messages
        self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)

        # Create a subscriber for odometry messages
        self.odom_sub = rospy.Subscriber('/odom', Odometry, self.odom_callback)

        # Initialize the current pose
        self.current_pose = None

    def odom_callback(self, msg):
        # Update the current pose
        self.current_pose = msg.pose.pose

    def move_distance(self, distance, speed):
        # Wait for the first odometry message to arrive
        while self.current_pose is None:
            rospy.sleep(1)

        # Get the initial position
        initial_position = self.current_pose.position

        # Create a Twist message for the movement
        twist = Twist()
        twist.linear.x = speed if distance > 0 else -speed

        # Publish the Twist message until the robot has moved the specified distance
        while self.calculate_distance(initial_position, self.current_pose.position) < abs(distance):
            self.vel_pub.publish(twist)
            rospy.sleep(0.1)

        # Stop the robot
        self.vel_pub.publish(Twist())

    def rotate(self, angle, speed):
        # Wait for the first odometry message to arrive
        while self.current_pose is None:
            rospy.sleep(1)

        # Get the initial orientation
        initial_orientation = tf.transformations.euler_from_quaternion([
            self.current_pose.orientation.x,
            self.current_pose.orientation.y,
            self.current_pose.orientation.z,
            self.current_pose.orientation.w])

        # Create a Twist message for the rotation
        twist = Twist()
        twist.angular.z = speed if angle > 0 else -speed

        # Publish the Twist message until the robot has rotated the specified angle
        while self.calculate_angle_difference(initial_orientation[2],
```

```
            tf.transformations.euler_from_quaternion([
            self.current_pose.orientation.x,
            self.current_pose.orientation.y,
            self.current_pose.orientation.z,
            self.current_pose.orientation.w])[2]) < abs(angle):
            self.vel_pub.publish(twist)
            rospy.sleep(0.1)

        # Stop the robot
        self.vel_pub.publish(Twist())

    @staticmethod
    def calculate_distance(position1, position2):
        return math.sqrt((position1.x - position2.x)**2 + (position1.y - position2.y)**2)

    @staticmethod
    def calculate_angle_difference(angle1, angle2):
        return abs((angle1 - angle2 + math.pi) % (2*math.pi) - math.pi)

if __name__ == '__main__':
    controller = MiR200Controller()

    controller.move_distance(-0.7, 0.2) # Move forward 1 meter at 0.2 m/s
    rospy.sleep(2) # Pause for a second
    controller.rotate( -1.5184, 0.3) # Rotate 90 degrees at 0.1 rad/s
    rospy.sleep(2) # Pause for a second
```

K | Automation process of chamfering - Python code

```python
#!/usr/bin/env python3

import subprocess

# MiR200 control - go to first target
subprocess.Popen(['python3','new_mir.py']).wait()

# Alignment of ee
subprocess.Popen(['python3','alignment_of_ee_with_checkerboard.py']).wait()

# Detect of pipes
subprocess.Popen(['python3','/home/toni/ws_moveit3/src/final_codes/Yolo/content/yolov5/detect.py']).wait()

# Chamfering of first cooler
subprocess.Popen(['python3','chamfering.py']).wait()

# MiR200 control - go to second target
subprocess.Popen(['python3','new_mir_2.py']).wait()

# Alignment of ee
subprocess.Popen(['python3','alignment_of_ee_with_checkerboard.py']).wait()

# Detect of pipes
subprocess.Popen(['python3','/home/toni/ws_moveit3/src/final_codes/Yolo/content/yolov5/detect.py']).wait()

# Chamfering of second cooler
subprocess.Popen(['python3','chamfering.py']).wait()

# MiR200 control
subprocess.Popen(['python3','new_mir_3.py']).wait()
```