Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Universal Teleoperation ROS Interface for Robotic Manipulators

vorgelegt von

Fabian Hendrik Wieczorek

MIN-Fakultät

Fachbereich Informatik

Arbeitsbereich Technische Aspekte Multimodaler Systeme

Studiengang: Software-System-Entwicklung

Matrikelnummer: 6911629

Erstgutachter: Prof. Dr. Jianwei Zhang

Zweitgutachter: Yannick Jonetzko

Abstract

In modern times, autonomous robotic systems play an increasingly important role, especially in industry. But some tasks are too complex to automate and require a human operator to control the robot which is called teleoperations. In this case, robots can be controlled directly or they support the operator with a certain level of autonomy. Teleoperations can also be used to teach robots more natural movements with learning by demonstration. In this thesis, a teleoperation ROS-based interface that aims to let the operator control robotic arms is presented. The interface is unspecific in terms of input device and manipulator as long as the hardware supports ROS. The operator can directly control the end-effector of the robot in Cartesian space. Two existing open-source approaches, namely jog_control and jog_arm, were analysed. The jog_control package was extended in its functionality to support absolute jogging in addition to relative jogging. The performance in terms of position error and time delay was evaluated with different tests. The results show that the simulated UR5 manipulator could be controlled without difficulties with time delays of around 0.35-0.5 seconds. The tests also revealed issues with infinite joints and collision avoidance in certain cases.

Zusammenfassung

In der heutigen Zeit spielen autonome Robotersysteme eine immer wichtigere Rolle, besonders in der Industrie. Allerdings gibt es Aufgaben, die zu komplex sind um sie zu automatisieren. Stattdessen übernimmt ein Mensch die Steuerung des Roboters, was als Teleoperation bezeichnet wird. In diesem Fall kann der Roboter sehr direkt gesteuert werden, oder den Benutzer mit einem gewissen Maß an Autonomie unterstützen. Teleoperation kann auch genutzt werden, um Robotern natürlichere Bewegungen durch learning by demonstration beizubringen. In dieser Bachelorarbeit wird ein auf ROS basiertes Teleoperation Interface untersucht, welches die Steuerung von Roboterarmen durch einen Operator ermöglicht. Das Interface ist unspezifisch im Hinblick auf Eingabegerät und Manipulator solange die eingesetzte Hardware ROS unterstützt. Der Benutzer steuert den Endeffektor mit kartesischen Koordinaten auf direkte Art und Weise. Zwei vorhandene Ansätze, mit den Namen jog_control und jog_arm, wurden dazu analysiert. Das jog_control Programmpacket wurde in seiner Funktionalität so erweitert, dass nicht nur relatives jogging, sondern auch absolutes jogging unterstützt wird. Die Performance der Erweiterung hinsichtlich Fehler in der Position und Verzögerungszeit wurde mithilfe verschiedener Tests evaluiert. Die Ergebnisse zeigen, dass der simulierte UR5 Manipulator ohne Schwierigkeiten gesteuert werden konnte und Zeitverzögerungen von etwa 0,35-0,5 Sekunden entstanden sind. Des weiteren zeigten die Tests, dass es Probleme mit infinite joints und in speziellen Fällen mit der Kollisionsvermeidung gibt.

Contents

List of Figures

1 Introduction

In recent years more and more domains started to benefit from robotics. Even though the world gets increasingly digitalized, a lot of tasks requiring physical manipulation or sensing remain. And some of those have a high potential being done efficiently with robotic support. Particularly in industry, robots offer great advantages when they are used for repetitive tasks. In most cases, those robots are designed to work autonomously. To achieve high precision or speed, they follow predefined routines specialized for the given job.

Yet, the major group of tasks are individual each time and thus, using robots for these hardly automatable jobs seems unfavourable at first sight. But there are numerous cases where robots could help humans in exceptional situations. Some of them can be found at hazardous workplaces. These are not a rarity and so there are lots of people being endangered by their job. Despite safety precautions, there are plenty of (fatal) occupational accidents every year [1]. Using robotic actors could prevent humans from having accidents in those hazardous environments. However, these robots would need to be operated by a person because developing an autonomous robotic system for each task is typically unfeasible in terms of time and costs. Teleoperated robots may be seen as a physical replacement for human attendance and not as a whole substitute. The task is being done by a person but carried out by a robot. This could save costs since robots can be kept general-purpose and more important, the person stays in a safe environment.

An example is a collapsing building. It is a possible consequence of an earthquake, an explosion, or other causes. Rescuers need to find out whether people have been buried underneath the debris. If that is the case, they only have a limited amount of time helping the victims. To save lives, rescuers sometimes need to tunnel themselves through the rubble to the victims, having very little space to move (Figure 1.1, left image). Because of further collapsing, this is a live risking act [2]. In this particular case, a teleoperated robot with similar crawling abilities (Figure 1.1, right image) may help to locate the victims and find a safe path for the rescuers.

When nuclear power plants or other nuclear facilities cease production, they need to be torn down. This process involves working with highly radioactive material since, in the end, the facility's original place should no longer emit serious amounts of nuclear radiation. Humans should be exposed to those materials as little as possible. In this context, robots can be of great advantage in different tasks such as moving contaminated material (Figure 1.2, left image) or examining risky objects and places (Figure 1.2, right image). Because of energy change, the number of nuclear decommissions will probably increase in near future.

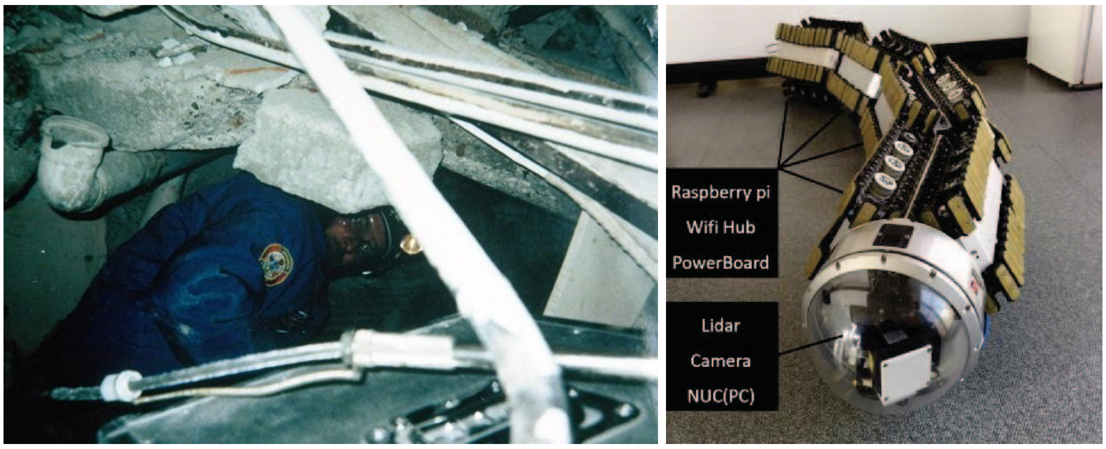


Figure 1.1: Hazardous workplaces are affecting peoples health in different ways. After an earthquake or an explosion people might get buried under the rubble and need to be found and saved by rescuers. **Left:** A member of the International Rescue Corps amongst debris[1]. **Right:** A teleoperated segmented rescue robot designed to work in those scenarios [3]. This robot can be used to find possible victims without risking the life of a rescuer.

But dangerous workplaces are not the only case of application for teleoperated robots. The exploration of the oceans becomes more important due to climate change and the growing demand for resources. Manned underwater vehicles are often expensive to ensure human security. Remotely operated vehicles can carry out complicated tasks while the operator stays on the water surface allowing longer and much safer operations. [6]

Sending teleoperated vehicles into space is a very powerful method to collect data from other celestial bodies than planet earth. Since the 1970s mars rovers were sent to the red planet for exploration missions. But since communication is very limited the control commands mostly consists of setting waypoints and behaviours. [7]

Not only can robots be physically at places being too dangerous for humans, but they can also be equipped with specific hardware that exceeds limited human skills. Such robotic systems are highly useful for example in surgeries, where robots can offer a high amount of geometric precision and less invasiveness. In this way, minimally invasive surgeries (MIS) can be carried out which do not require to open the patient's body. In MIS the surgeon works with long instruments through small incisions. This method is usually better for the patient's health than open surgery, but this method also has some disadvantages: (1) Working with those instruments is not intuitive since they cannot be moved freely, (2) the surgeon cannot see the environment directly and (3) the contact forces can hardly be sensed. Since robotic manipulators can be equipped with a variety of sensors, they could help surgeons by precisely executing commands while

[1]Source: http://www.intrescue.info/hub/index.php/missions-2/, Accessed: 14.02.2020

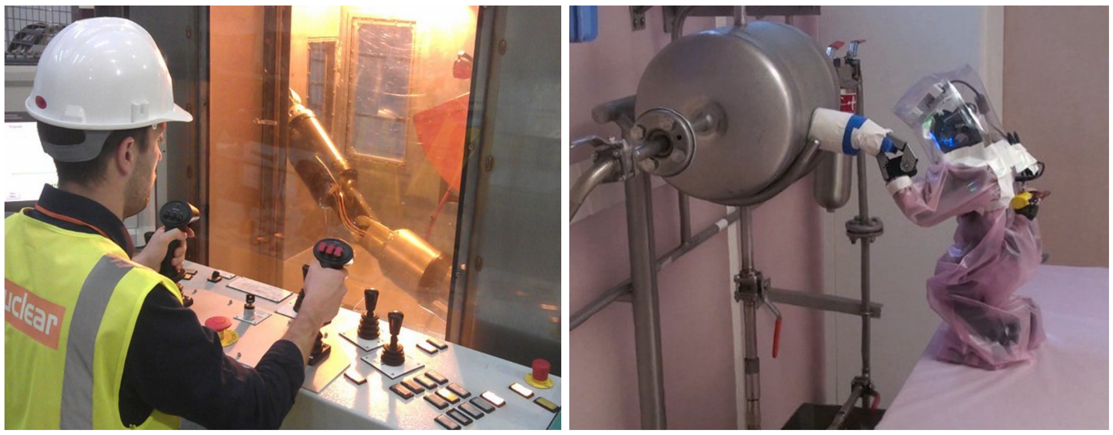


Figure 1.2: Nuclear decommission involves working with highly radioactive material. Letting humans carry out this work directly should be avoided as much as possible. **Left:** A teleoperated 6 DOF robotic arm with a payload of 100kg controlled by two joysticks [4]. It can be used to move contaminated material while the operator stays at a safe distance. **Right:** A DARwIn-OP was sent to take smears and assess object contamination [5].

providing feedback on the resulting interaction forces. These advantages safe valuable time in the operation room and also lead to a better overall quality of the surgery [8]. But teleoperated robots also appear in everyday life such as RC toys in backyards or as claw cranes [9] at funfairs.

In 2013 the DARPA (Defence Advanced Research Project Agency) [10] started their DARPA Robotics Challenge to promote the development of teleoperated ground robots that can execute complex tasks in hazardous scenarios [11]. These tasks require a lot of different skills including driving alone, walking through rubble, turning a valve, etc. [12]. The exercises are barely feasible without the human-in-the-loop model showing what can be possible when combining robots and human operators.

Teleoperation is an abstract term. It includes all kinds of remote-controlled robotics varying in multiple aspects such as connectivity, autonomy and complexity. A teleoperated robot may be controlled via the internet over multiple continents but it can be also wired to the control unit being in sight distance to the operator. The frequency of commands can be diverse as well. A Mars-rover, receiving perhaps only one command a day, needs a high level of autonomy whereas some real-time robotic systems require the user to control it directly. Vehicles being able to navigate in four directions can be teleoperated as well as stationary robotic arms with 8 degrees of freedom.

This work addresses teleoperation methods for articulated robotic arms, called manipulators, with at least 6 degrees of freedom (DOF) and exactly one end-effector (Figure 1.3). The aim is to have the end-effector controlled directly by the user. This means autonomous or semi-autonomous mechanisms are not considered.

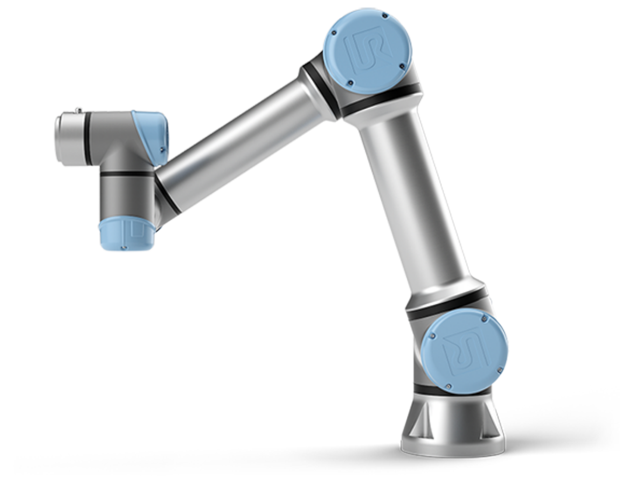


Figure 1.3: The UR5 robot arm from Universal Robots [13]. It has 6 degrees of freedom and one end-effector. The goal of this work is to have an interface allowing teleoperations on diverse manipulators like this one.

The general principle of such a system is shown in Figure 1.4. A teleoperated manipulator continuously receives input data from the user and processes it to interact with its environment. As a consequence, the robotic arm moves its links a little according to the input. After that, the user needs to be informed about the new state by receiving certain feedback from the robot. This can be for example visual feedback, haptic feedback or a digital representation. Based on this feedback the user adjusts the input and the procedure starts over again. Because the manipulators in this work will be controlled directly, this loop needs to happen with a certain frequency otherwise the delay between input and execution would make the control too difficult to use.

But this approach also leads to some problems since humans have different limitations than computers have. The teleoperation approach would need to take simple input and convert it to commands being performed by complex hardware. For robotic arms, this input usually comes in form of the end-effector position and orientation (its pose). Every joint need to be set according to the desired pose. This conversion from Cartesian space to joint space can be quite expensive in terms of computational resources. However, the processing and acting need to happen without larger delays because they would lead to impreciseness and loss of control.

Another aspect is the input device used to control the robot. Input devices vary in form and functionality (Figure 1.5) and the right choice depends on the type of application. They are used as human-computer interface allowing the user to control the end-effectors pose. Mainly, this can be done in two different ways: Relative transformation would tell the end-effector how to change its current state and absolute positioning tells the end-effector a goal it needs to reach. Both mechanisms need to be considered, otherwise, the system may lack support for special types of input devices.

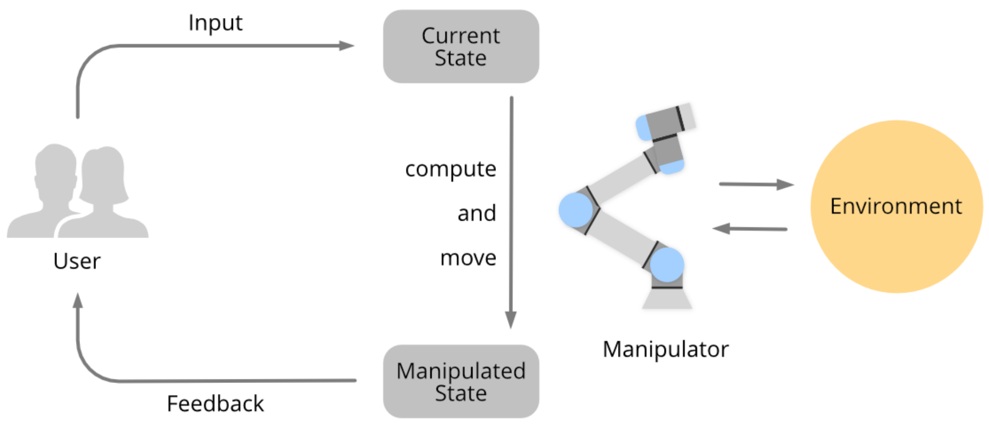
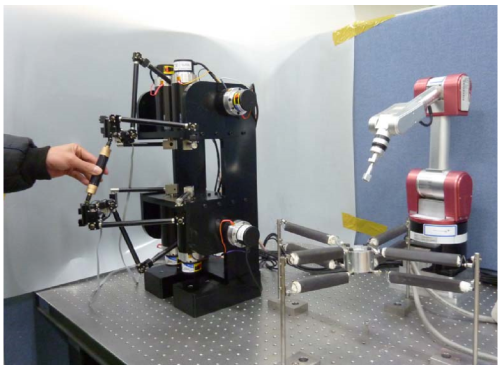


Figure 1.4: A visualization of the *human-in-the-loop* model. Teleoperated robots (right) are controlled by taking input from the user (left) to interact with their environment. By processing the input, the robot moves its joints according to the user's commands. Being the operator, the user needs to supervise the situation and the robot's actions. Hence, the robot needs to provide feedback allowing the user steady controlling.

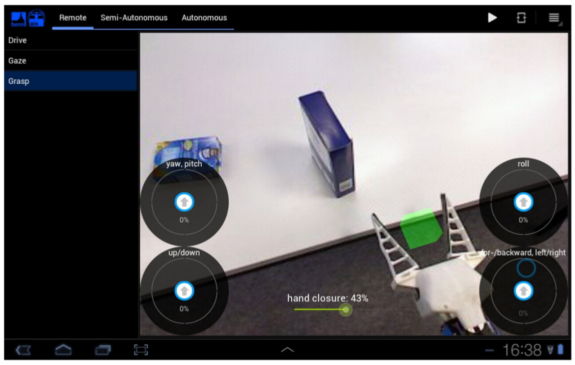
Technical aspects play a significant role in teleoperation systems. Robotic actuators (e.g. servomotors) can be controlled in many different ways. A meaningful choice of controllers affects the quality and usability of the system. Software arrangements have to be made to prevent unexpected behaviour on behalf of the robot. Another issue in robotics is the broad diversity of hardware. This complicates the development of robotic applications, due to the need of hardware-specific software implementations. With the idea to increase the use of teleoperation systems, these integration problems need to be taken into account.

The goal of this work is to propose a universal teleoperation interface based on ROS. This interface should be applicable for most types of manipulators requiring only little configuration. Furthermore, no type of input device will be primarily addressed. Instead, the focus is on the creation of an interface usable by a large number of input devices without much effort. To achieve this, two existing approaches will be analysed and one will be adapted to meet certain criteria.

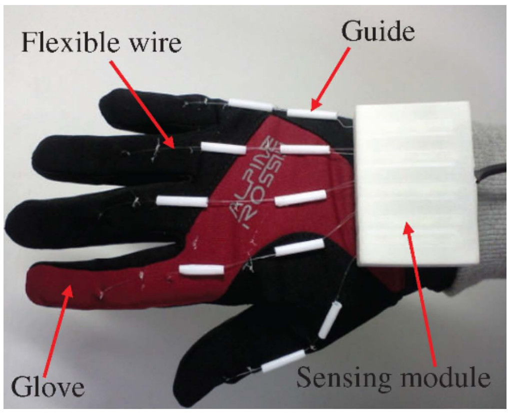
Having the motivation and the goal of the thesis stated, this work will continue with presenting different teleoperation systems along with different use cases in the related work. In chapter 3 the necessary basics for this work will be explained. Going towards a solution, chapter 4 takes a look at two existing approaches. One approach will be extended in chapter 5. Chapter 6 will examine the proposed interface based on the stated methods. After that, the outcoming results will be discussed in chapter 7 and a final conclusion will be stated in chapter 8 giving also a brief review of this work.



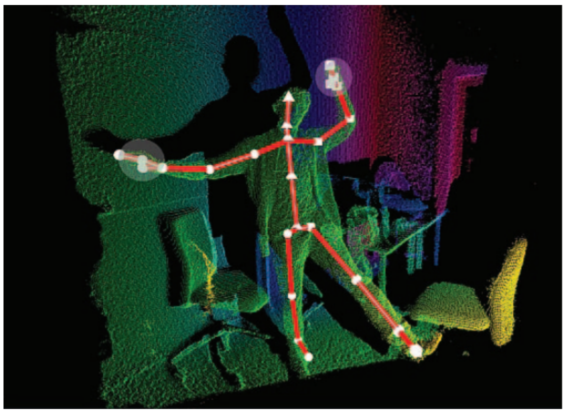
(a)



(b)



(c)



(d)

Figure 1.5: The input device plays a fundamental role in direct robot control. Four different input methods are depicted. **(a)** A 6 DOF haptic input device determines the pose of the robot's end-effector. It also simulates the environment forces that the robot senses as additional feedback. [14]. **(b)** A joystick-like graphical user interface displaying an image of the scene. It also supports high-level commands such as grasping a selected object [15]. **(c)** A wearable glove sensing the fingers states. The position is determined with a set of IMUs placed on the human's brachial joints [16]. **(d)** Skeleton tracking using the Microsoft Kinect [17].

2 Related Work

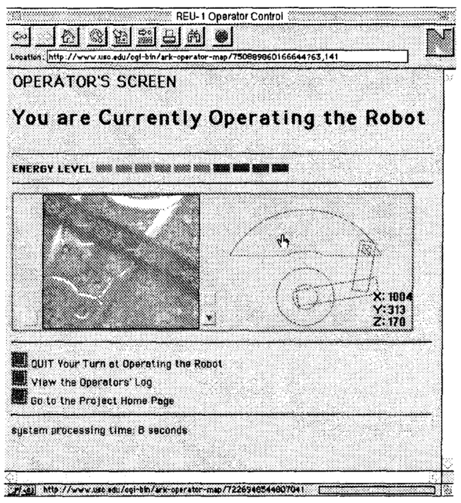
There are numerous different applications for teleoperated robots. Some examples were already given in chapter 1 showing that teleoperated robotic systems vary in form and functionality. This chapter summarizes different research publications featuring these kinds of systems and scenarios for application.

Service robots in the domestic environment are typically used to assist humans with everyday tasks. Muszynski et al. developed a teleoperation system based on ROS for the Cosero and the Dynamaid service robot utilizing different levels of autonomy [15]. Driving, gazing and manipulation skills can be executed autonomously, semi-autonomously and user-controlled. As input device, a handheld computer running Android OS was used. A joystick-like UI was used to control the robot directly (Figure 1.5, b). Furthermore, the robot was taught different skills like grasping selected objects or navigating through the interior. These skills could be combined to fully automate certain tasks.

With the advancements of the internet, communication around the globe was revolutionised. In September 1994 the first teleoperated robotic system, controllable via the world wide web, went online. This project was intended to show the feasibility of remote-controlled robot systems over the internet. Using the HTTP protocol, users could connect to the system and control the manipulator. A greyscale video stream and a visual depiction of the robots state gave feedback to the user (Figure 2.1, left image). The latter also served as the control unit. With this setup, users could search for artefacts buried in sand, by manipulating the remote environment with compressed air. [18]

These systems can be used especially in dangerous environments where complex manipulation is necessary but human intervention not possible because of risks to health. A practical example has been done by Ma et al. using teleoperated robots to work with radioactively contaminated material [19]. A system of four manipulators (two 6-DOF stationary and two 9-DOF mobile robots) was proposed. These robots were teleoperated by three master sides equipped with video feedback, a distributed virtual environment, a head-mounted display and haptic device for 3D interaction. The task was to replace a fuel rod of a nuclear plant by three human operators via the internet (Figure 2.1, right image). All operators performed the work safely and efficiently, showing the potential of teleoperated robotics in groups.

With the Microsoft Kinect Marinho et al. developed a control for the Schunk 9-DOF LWA which uses simple arm gestures and commands. This approach intends to perform pick and place operations which do not require force feedback. Since the user needs to adjust it's own hand position based on visual feedback, the robot needs to be placed in sight distance. Even though the end-effector could not be manipulated


REU-1 Operator Control

Location: http://www.usc.edu/cgi-bin/ark-operator-map/75088986016664?63,141

OPERATOR'S SCREEN

You are Currently Operating the Robot

ENERGY LEVEL

X: 1004
Y: 313
Z: 170

QUIT Your Turn at Operating the Robot
View the Operators' Log
Go to the Project Home Page

system processing time: 8 seconds

http://www.usc.edu/cgi-bin/ark-operator-map/7226946544807041

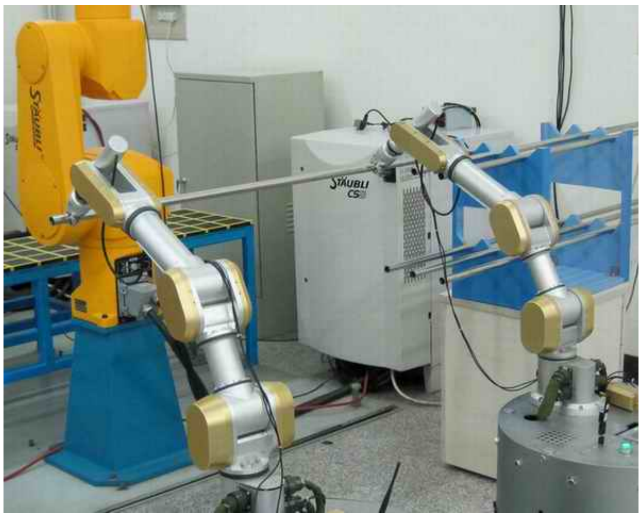


Figure 2.1: **Left:** The user interface of the first internet teleoperated robotic system [18]. It shows the greyscale video stream and a representation of the robots current state that can be used to give commands. **Right:** Two mobile, teleoperated robots carry a fuel rod given by the larger robot in the background [19]. The robots were controlled by three individual users over the internet.

in its orientation, the experiment showed that a complex robotic arm can be controlled naturally by simple hand gestures. [20]

With the advancements of Virtual Reality (VR), especially Head-Mounted Displays (HMD) like the Oculus Rift or the HTC Vive, the type of robot feedback has reached new dimensions. Now, users can observe the scene from the robot's point of view in 3D while controlling the robot [21, 17]. Whitney et al. have developed a framework for ROS integration in the Unity Game Engine utilizing VR. The framework consists of a set of scripts, connecting the Unity Scene to the ROS master via a Rosbridge implementation. Its main purpose is to represent the robot's point of view via point cloud in Unity (Figure 2.2) and to utilize the position tracked controllers as input device [22]. The effectiveness of using VR with an HMD in teleoperation tasks was evaluated with a cup stacking task being performed by 18 participants each. It showed that using the VR interface the task was completed 66% faster on average in comparison to the conventional keyboard and monitor interface. Furthermore, a lower workload and higher usability were measured. [23, 22]

Other approaches have made use of electromyographic (EMG) signals. These signals come from skeletal muscles and they can be measured by sensors applied to the skin surface. Artemiadis and Kyriakopoulos proposed a methodology which can not only estimate the position and motion of limbs but also the forces applied by the user to the environment [24]. The system consists of three low-frequency position tracking sensors mounted on the user's shoulder and elbow points. Four EMG sensors recorded the activity of four muscles of the shoulder and elbow joint. Even though position and motion

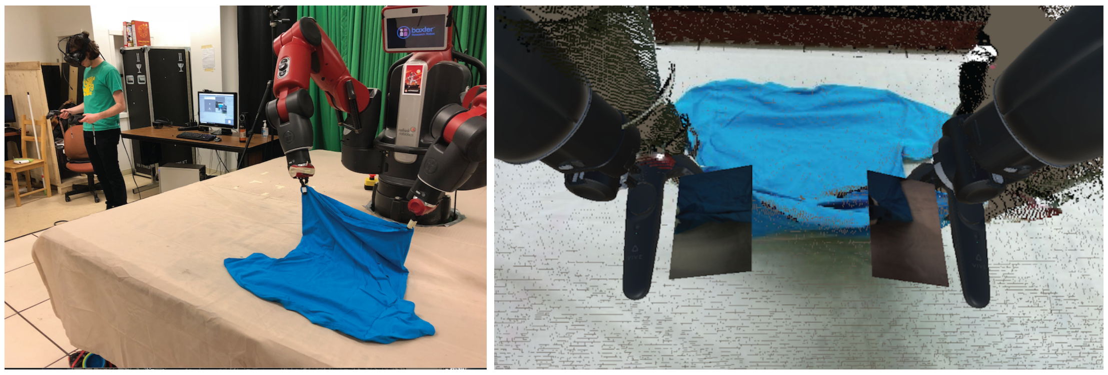


Figure 2.2: **Left:** The user controls a Baxter using virtual reality and ROS Reality. The task is to fold a shirt. **Right:** The users view through the VR headset. It shows a combination of a point cloud from the scene and a mesh model of the robot. [22]

were geometrically restricted to a plane, a 7-DOF manipulator (PA-10, Mitsubishi Heavy Industries) was successfully controlled with high accuracy. Similar controls without the need for laboratory equipment can be achieved with the Myo armband (Figure 2.3, left image). This wearable device has not only EMG sensors but is also equipped with a gyroscope, an accelerometer, and a magnetometer and it communicates wirelessly over Bluetooth. By utilizing these instruments, Çoban and Gelen successfully controlled a 6-DOF manipulator (Mitsubishi RV-7FL-D) with a gripper (Figure 2.3, middle/right image) to perform pick and place operations. [25]

A state-of-the-art system for teleoperations with haptic feedback was recently introduced by Haptx. The *Tactile Telerobot* [26] consists of two haptic gloves developed by the company itself, two Universal robots and two Shadow Dexterous Hands, each equipped with BioTac Sensors on their fingertips. The users pose and motion of both arms and hands are directly imitated by their robotic counterparts. Providing accurate haptic feedback for each fingertip, the system offers very precise and intuitive control allowing to do tasks that require high amounts of accuracy. Despite being interesting for this work, this approach cannot be taken into consideration since it was of commercial use and neither source code or research publications have been published.

An interesting approach using a high level of autonomy was developed by Pruks et al. targeting nuclear plant robotics. A teleoperation system where mobile robots follow a path sketched by the user (Figure 2.4) was proposed [27]. The robot delivers visual feedback of the room it faces. On this image, the user finds the best fitting path and sketches it with a special input device. The robot then interprets the sketched path and follows it autonomously. Depending on the situation, this shared autonomy approach performed better than fully autonomous or fully user-controlled mechanisms.

Modern medicine without the assistance of robotic systems is no longer imaginable. Since the mid-1980s a wide field of different medical robots has been developed [28]. These systems can offer quite a lot of advantages including high precision, higher qual-

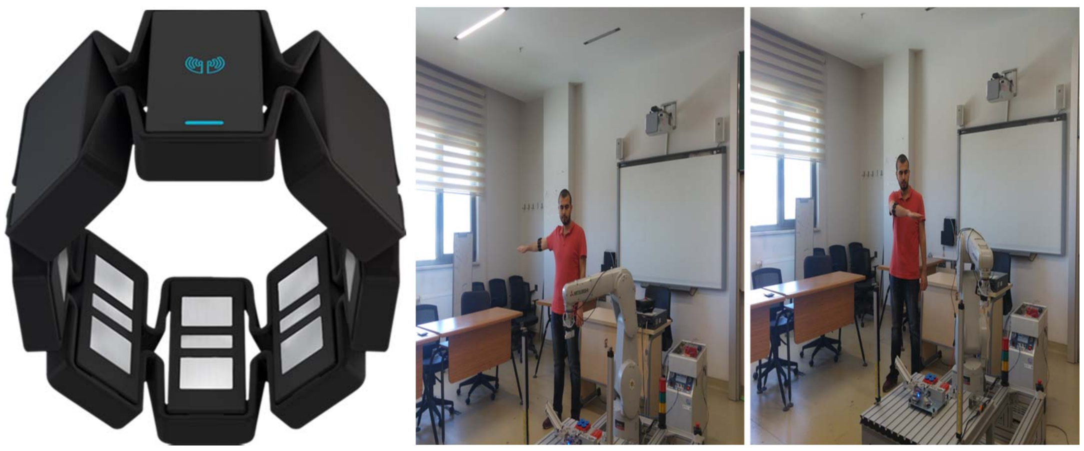


Figure 2.3: **Left:** The Myo armband leveraged to operate a 6-DOF manipulator. It is used to measure electromyographic signals from the user. Then, the data gets transmitted wirelessly over Bluetooth. **Middle/right:** The operator, wearing the Myo armband on his right arm, controls the robot wirelessly by only moving his arm. [25]

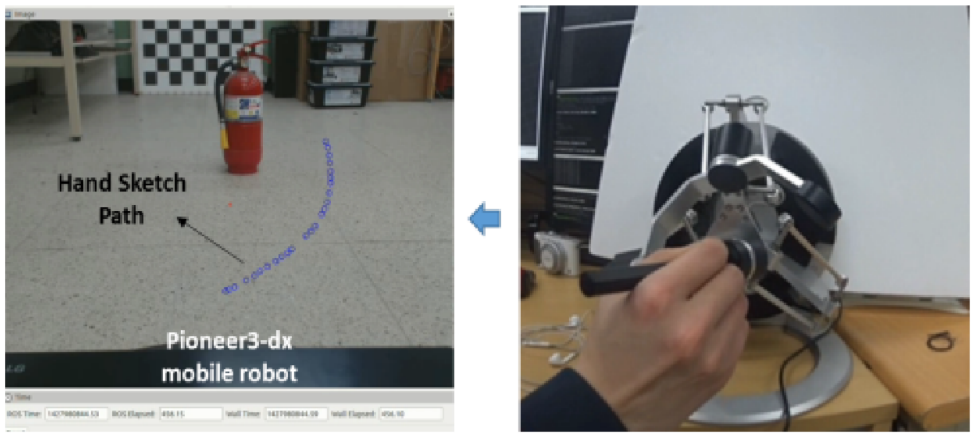


Figure 2.4: Teleoperating a mobile robot based on shared autonomy. **Left:** The robot gives visual feedback of the scene of its point of view. **Right:** Based on the image, the user then sketches a path with a special input device. This path gets interpreted by the robot and it follows it autonomously. [27]

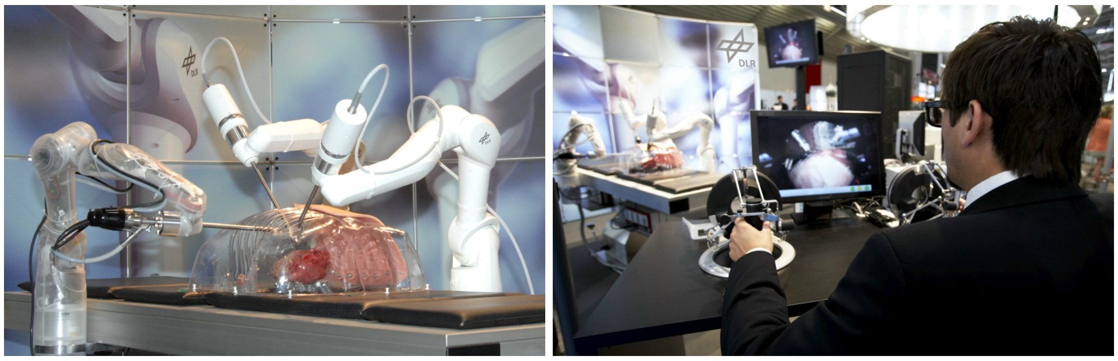


Figure 2.5: **Left:** Three robotic arms with each 7 degrees of freedom help in minimally invasive surgeries. They are equipped with two surgical instruments and one stereo endoscope. Force torque sensors on the surgical instruments help to give feedback to the surgeon. **Right:** The operator controls the robotic system with two input devices. The stereo display provides visual feedback coming from the endoscope. [31]

ity and the need for less valuable time in certain interventions [8]. The most interesting robotic systems for this work are teleoperated, semi-autonomous robots which serve as tools for human surgeons.

A good example of how micro-robotics can increase precision and quality in some interventions are nonholonomic steerable needles. Needle insertions are important for many clinical procedures. But in some cases, the target area for the needle tip might be unreachable without avoiding obstacles such as nerves, vessels and bones. A common praxis is to use a flexible needle with an asymmetric tip. Due to the resistance during insertion, the needle will bend and the direction can be controlled through rotation of the needle [29, 30]. This method requires to take action in joint space and is therefore not intuitive for humans. Robotically steered needles can be controlled in Cartesian space by defining the target positions of the needle directly. To place the points, a special haptic device is used. This successfully tested teleoperations approach leads to smaller insertion length and less targeting errors. [30]

In minimally invasive surgeries (MIS), assisted robotics have promising potential. Tobergte et al. developed a teleoperation system for such surgeries. The setup consists of three 7-DOF manipulator equipped with surgical instrument (Figure 2.5, left image) and two haptic input devices (Figure 2.5, right image). The usage of the system requires a lot of planning based on a virtual environment and the patient's data. During the actual intervention, the surgeon can intuitively control the surgical devices and senses direct feedback resulting from his operations.

3 Fundamentals

In this chapter the basic knowledge necessary to understand this work will be covered. It starts with section 3.1 giving an overview of coordinate frames that are fundamental to understand the representation of the joints position and orientation. Followed by section 3.2 outlining the differences between the joint space and Cartesian space and how they are used to control serial chains of joints. The concepts of switching between the two spaces are referred to as forward and inverse kinematics and will be covered in section 3.3. Making use of forward and inverse kinematics, jogging, a form of robotic movement where the moving directions are communicated during execution, will be explained in section 3.4. Section 3.5 takes a look at robotic controllers that are responsible for driving the robot's joints accordingly while section 3.6 briefly explains the popular robotics framework ROS. At last, section 3.7 takes a look at the motion planning framework Moveit! that is going to be used for the jogging implementation.

The simplified process of a teleoperation system is displayed in Figure 3.1. It is split into different tasks (A-E) that are carried out by different actors (coloured boxes and user). It starts with the user giving a command with the help of an input device (Figure 3.1, A). This device can be for example a joystick, an IMU, a graphical user interface, or a pose tracking system. The command tells the robot to move its end-effector in a certain way. Depending on the input device this can either be a moving direction or the desired pose.

The command will then be received by the teleoperation interface using the input to determine the end-effectors new pose (Figure 3.1, B). After the pose was determined, the new joint positions need to be calculated and published to a suitable controller (Figure 3.1, C).

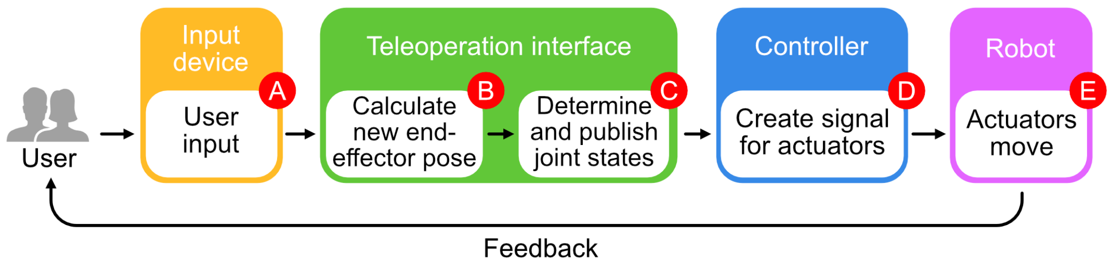

User
Input device
User input
A
Teleoperation interface
Calculate new end-effector pose
B
Determine and publish joint states
C
Controller
Create signal for actuators
D
Robot
Actuators move
E
Feedback

Figure 3.1: Overview of the teleoperation process loop. The necessary tasks (A-E) are carried out by different actors (coloured boxes + User).

The controller takes care of the actuators to move accordingly. When the earlier published joint states are received, it checks the error between the joints actual state and the desired state producing a signal so the actuators move properly (Figure 3.1, D). As a result, the robots joints will move so that the end-effector follows the user's input (Figure 3.1, E). At last, the user receives certain feedback from the robots new state. This can be for example visual feedback but there are also devices providing the user with haptic feedback.

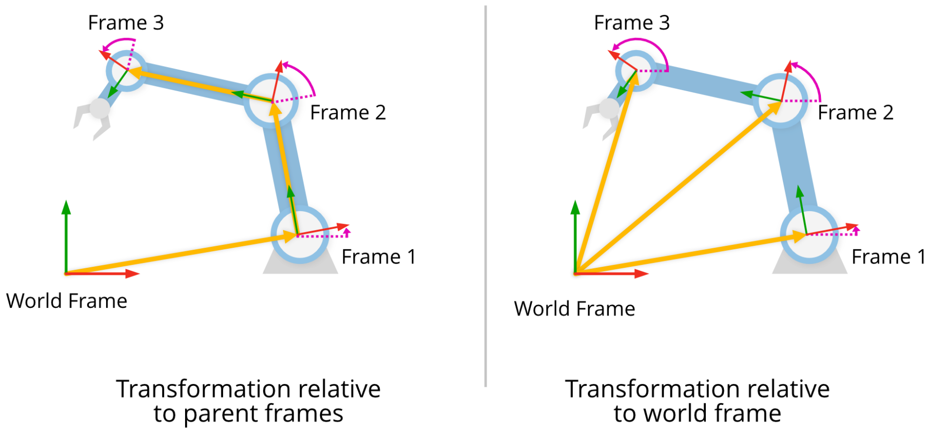

Frame 3
Frame 2
Frame 1
World Frame
Transformation relative to parent frames
Frame 3
Frame 2
Frame 1
World Frame
Transformation relative to world frame

Figure 3.2: **Left:** The frame of a link (red/green arrow pairs) is always relative to its parent frame. That means the Cartesian displacement (yellow arrows) and the rotational displacement (magenta arrows) have their origins at the parent's frame. **Right:** The links poses are described in world coordinates so that all the origin of the links frames is the world frame.

3.1 Coordinate Frames

The spatial state of a body such as the robot's links is represented with coordinate frames (or simply frames). A coordinate frame is defined by three orthogonal basis vectors with all the same position origin. This origin and the orientation of the vectors make up the pose of a frame. It is important to notice that a frames pose is always relative to another frame as shown in Figure 3.2 (left image).

Frames play an important role when applying the users input to the end-effector (Figure 3.1, A + B). Since the user stays in a fixed frame (e.g. the world frame) the end-effectors pose needs to be determined relative to the same frame as visualized in Figure 3.2 (right image). Converting the end-effectors pose so it is relative to for example the world frame is known as forward kinematics and described in section 3.3. [32]

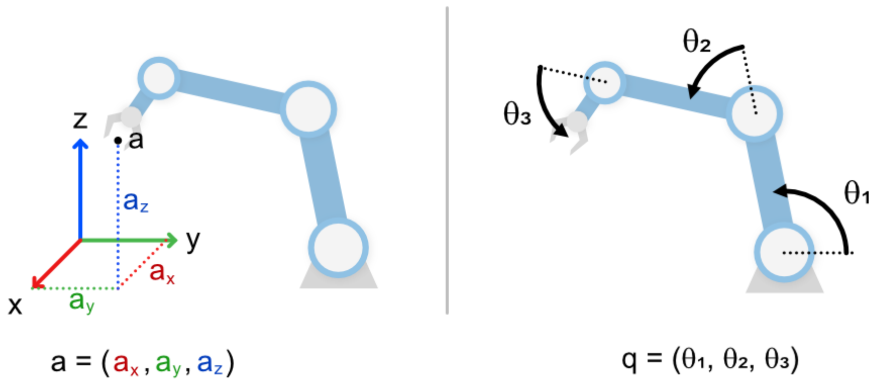


Figure 3.3: **Left:** A visualization of the Cartesian space with a point a. **Right:** A visualization of the joint space for 3 revolute joints with a configuration q.

3.2 Cartesian Space versus Joint Space

In robotics, there are two major types of spaces that are frequently used: The Cartesian space, and the joint space. The **Cartesian space** is defined by three orthogonal axes where each point is described by a 3-tuple of numbers, one for each axis. The intersection of the three axes is called the origin. Figure 3.3 (left) shows a point a defined by its three coordinates (a_x, a_y, a_z). [33]

The **joint space** is an n-dimensional space where n equals the number of the robot's joints and a point in this space corresponds to a configuration of the robot. One configuration can be described as a set of the joints coordinates. So for an all-revolute robot, this set consists of the joints angles. Figure 3.3 (right) illustrates a robot with three joints where the joints angles are defined as θ_1, θ_2, and θ_3 so that the configuration q of a robot is defined as $q = (\theta_1, \theta_2, \theta_3)$. [34]

The robot's joints cannot be moved in Cartesian spaced directly. Instead, they are controlled in joint space. In the context of teleoperations, this is a problem because humans are not used to operating in joint space. This means the input of the teleoperation interface (Figure 3.1, A) is defined in Cartesian coordinates while the output (Figure 3.1, C) is a configuration of joint coordinates. The necessary conversions between Cartesian space and joint space are achieved with forward and inverse kinematics which is explained in section 3.3.

3.3 Forward/Inverse Kinematics

Forward and inverse kinematics can be used as conversion between joint space and Cartesian space.

Forward kinematics (FK) refers to the conversion from joint space to Cartesian space where the joints positions are given and the pose of the end-effector will be determined.

Inverse kinematics (IK) refers to the conversion from Cartesian space to joint space where the end-effectors pose is given and the joints positions will be determined.

The Forward kinematics problem is very straight forward. The link states and their respective geometry is known. The goal is to calculate the transformation between the robots first frame and its last frame. This is done by concatenating all transformations between the frames of adjacent links starting with the base frame. For a 6-DOF chained manipulator, this transformation would be

$$^{0}T_{6} \;=\; ^{0}T_{1}\,^{1}T_{2}\,^{2}T_{3}\,^{3}T_{4}\,^{4}T_{5}\,^{5}T_{6} \tag{3.1}$$

where $^{0}T_{1}$ is the transformation from the base frame (frame with index 0) to its successor based on the joints position. The outcome is the pose of the frame with index 1 relative to the base frame. Further concatenating the results ends in acquiring a transformation from the base frame to the end-effectors frame ($^{0}T_{6}$). [32]

In teleoperations, forward kinematics is used to determine the end-effectors pose in Cartesian space so the user input can be applied to it (Figure 3.1, B).

For the inverse kinematics problem, only the base frame and the desired pose of the end-effector relative to the base frame is given. This corresponds to the part in teleoperations where the new end-effector pose is known in Cartesian space and the joint coordinates need to be determined (Figure 3.1, C).

But not every given pose will result in a solution. For example, if the desired pose lies not within the robots reachable space then no solution exists. It is also possible for some given poses that multiple solutions exist. Another aspect to mention is singularities. A singularity occurs when two or more of the robot's links line up parallel so the angle between them equals 0° or 180° making the robot redundant. In this case, it loses degrees of freedom making further movements difficult. [35, 32]

There exist multiple approaches to solve inverse kinematics which can be divided into closed-form solutions and numerical methods. The closed-form solutions are faster than numerical methods but robot specific. The numerical methods are computational more expensive and sometimes fail to find all possible solutions but can be universally used. [32]

3.4 Jogging

The concept of jogging is to manually control the robot by making small but very frequent changes in its state [36]. The result is a fluent movement dirigible in real-time. In this work, jogging will be used to continuously determine the end-effectors pose by the user's input. Before the robot moves, the state needs to be calculated in advance.

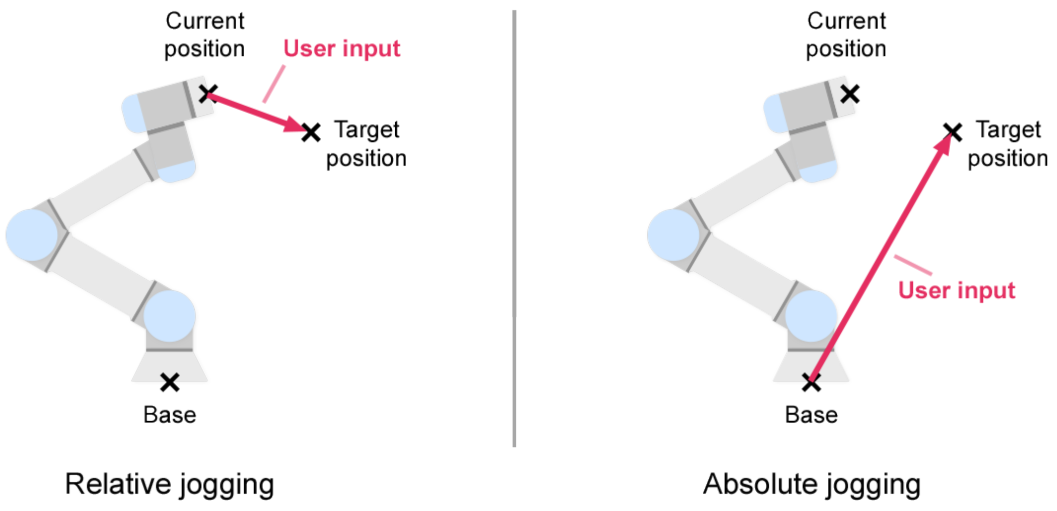


Figure 3.4: Depending on the input device, different methods for jogging may be considered. **Left:** Relative jogging means the user gives the robot commands relative to the end-effector frame. In other words, the user tells the end-effector to move in a specific direction. This can be useful for input devices having little action space for example joysticks. **Right:** Absolute jogging means the user gives commands relative to a fixed frame (in this case the base). In this case, the end-effector is commanded a new pose independent of its current state. This can be used with for example position tracking systems.

This is due to the necessary conversion from Cartesian space to joint space (see section 3.3). Calculating the end-effectors new state depends on the input devices that are used (Figure 3.1, B). There are two methods mainly differentiating in the type of input that is needed:

Absolute Jogging: The poses current values will be replaced with the new values defined by the user input (Figure 3.4, right image). This applies to the position coordinates as well as the orientation quaternion. The new position and orientation values do not depend on the current ones. This method requires knowledge about the robots reachable space because setting an invalid pose (e.g. too far away) results in an IK error since no solution exists and thus the robot will not move. An analogy would be the steering wheel in a car which directly controls the orientation of the wheels. This jogging type can be used with input devices such as data gloves, pose tracking systems or 3D cursors.

Relative Jogging: The new position and orientation values are determined by the current values and an offset which is the user input. The offset values will be added to the current values (Figure 3.4, left image). This method requires only information

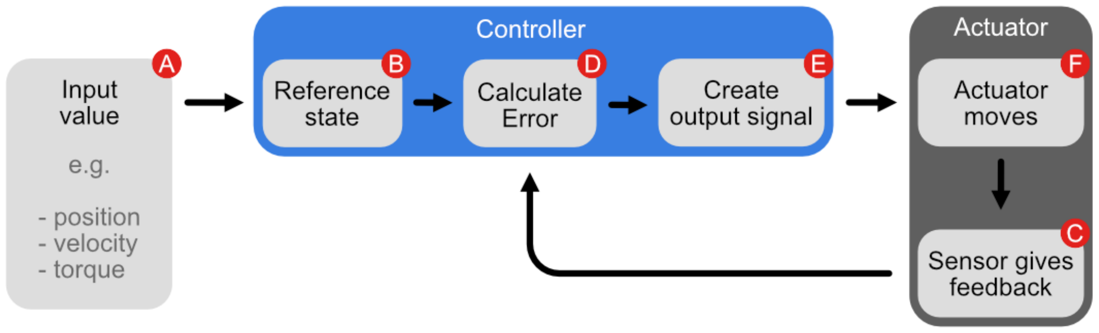


Figure 3.5: A controller is part of a closed-loop system operating actuators (e.g. servomotors). The controller updates its reference state every time a new input value (e.g. position, velocity or force) is given (A). The reference state (B) and the actuators actual state (C) are used to calculate the error between the desired and actual state (D). With this error, an output signal can be created (F) that is used to drive the actuator (F). Changes in its state are measured by a sensor providing the previously used feedback (C).

about the magnitude of the values since the robots current pose is automatically in its reachable space. It can be used with input devices such as IMUs (Inertial Measurement Units) or joy-sticks.

3.5 Controllers

A controller is a mechanism that controls an actuator by collecting feedback of its current state and comparing it to the desired state [37]. The actuator can be for example a motor, a heater, an electromagnet, or anything that takes electrical signals as input and changes its physical state based on these signals. In this work, the actuators that are going to be controlled are the servomotors of the robots.

Controllers are part of a closed-loop system which is illustrated in Figure 3.5. The controller needs to be given an input to update its reference state (Figure 3.5, A). For rotary servomotors, this can be for example an angle position, a velocity or a torque. The reference state (Figure 3.5, B) is used together with the actual state of the actuator to calculate the error (Figure 3.5, D). This state is measured with a sensor attached to it providing the controller with feedback (Figure 3.5, C). This feedback can also be more than one value in case multiple sensors are attached. After that, the error is used to determine the output signal (Figure 3.5, E). How the output signal is calculated depends on the implemented controller logic. The signal is then used to drive the actuator accordingly (Figure 3.5, F). [38]

In the context of teleoperations, the input value is typically the joints position. But different types of feedback can be used to create varying behaviour. Commonly used is position feedback forming a *Position Controller*. This controller drives the corresponding joints into the desired position with predefined force. If the force is rather high, the robot can potentially cause damages when collisions with its environment occur. This is because the controller is unaware of the robots surroundings.

An *Impedance Controller* uses not only position feedback but also force feedback. This controller controls the applied force by setting it in relation to the position error which is called mechanical impedance. The resulting behaviour can be compared to a virtual spring between the end-effector and the desired position: larger position errors will result in more applied force. This means collisions with the environment are not dangerous at small position errors since the applied force is decreased. [38]

3.6 ROS - The Robot Operating System

The Robot Operating System (ROS) is a set of software libraries and tools to help to develop robotic applications [39]. Being an open-source project, ROS is widely used by researchers, hobbyists, and industry. Its modular architecture is designed to make code more reusable. ROS was created because developing an application for robots can be quite demanding. The main reason for that is the wide field of diverse hardware. Without standardisation in software, the reuse of code is mostly difficult. That means changes in hardware often leads to a reimplementation of the functionality. To have an application running on a robot, a large codebase is often needed containing, for example, motor drivers, input processing and high-level application software. Rewriting this code prevents fast development especially when there is little knowledge in terms of the technical aspects [40]. ROS counters these issues and therefore it simplifies and accelerates the whole software development process. Being no actual operating system, ROS runs on Unix-based systems primarily on Ubuntu and Mac OS X [41, 42, 43]. In this work, the teleoperation interface will be developed based on ROS. Only the parts necessary to understand this work will be covered.

3.6.1 ROS Overview

The most important part of the system is the ROS master. It is the junction point connecting all nodes and services as well as providing resources such as parameters. A node is an independent process performing any kind of computation. They are meant to have well-defined purposes so the system is built upon many nodes. For example, one node could control a camera taking images of the environment, another one would do the image processing, one node could be responsible for navigation, one would control the robot's wheels and so on. [44, 45]

It is essential for nodes to communicate with each other. To do so, ROS provides communication over topics, services and action services allowing nodes to exchange messages with each other. A message is a data structure containing typed fields such as primitives, arrays and other messages. Alongside a set of predefined message types, new types can be defined with a special `msg` file.[46]

High-level message types being used for publish/subscribe matters should contain a `header` field. This field contains metadata describing the chronological order as well as information about the origin frame. A brief overview is given in Table 3.1.

Name	Type	Short Description
`seq`	`uint32`	Arranges the messages of one node into chronological order.
`stamp`	`time`	Stores the current uptime of the robot at the moment the message is created.
`frame_id`	`string`	The id of the frame which the messages data refers to.

Table 3.1: The content of the Header message type with a short description for every field. This data structure is often used in top-level message types.

A topic is a named bus that is used to broadcast messages from one node to many others. To receive messages, nodes can subscribe to one or multiple topics. Other nodes can publish their messages to the relevant topics. These messages will then be forwarded to the nodes having a subscription to the respective topic. The message transport is based on either TCP/IP (Transmission Control Protocol/Internet Protocol) or UDP (User datagram protocol). Topics anonymise the publish and subscribe mechanics to decouple message generation from consumption. [47]

While topics are only for one-way message transport, ROS services are used for request and response communication. A node offers a service under a name and needs to define two messages. A request message and a response message. The service can be used by other nodes similar to a simple procedure call. [48]

3.6.2 ROS Controllers

In ROS, controllers (see section 3.5) are implemented in the `ros_control` framework. Originally it was developed based on the `pr2_controller_manager`, a controller framework for the PR2. But now, `ros_control` is robot unspecific aiming to support reusability. In ROS, controllers serve as interfaces for the application (Figure 3.6, yellow boxes). The communication happens with ROS messages over topics or services (Figure 3.6, turquoise boxes). Each controller has a well-defined state (loaded, unloaded and running) and is managed by the Controller Manager (Figure 3.6, light blue box). ROS controllers are not limited to control a single actuator with a single property. Instead, complex mechanisms can be implemented controlling a whole group of joints. Over time, a lot of ready-to-use controllers have been developed and if necessary, custom controllers can be built as well. [49, 50]

ROS controllers can be reused because they do not have direct access to the hardware. The communication to actuators and sensors is managed via *hardware interfaces* (Figure 3.6, red box). These interfaces can perform actual `read/write` actions via different communication buses such as Serial, USB or Ethercat. In opposite to controllers,

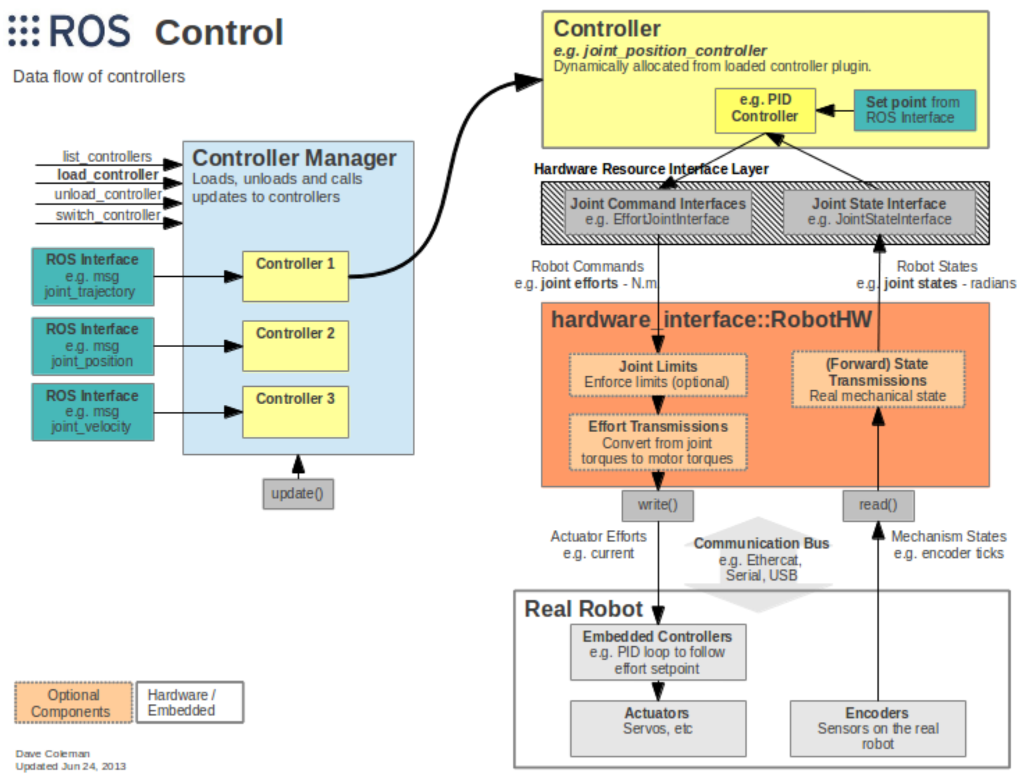

ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin.
e.g. PID Controller
Set point from ROS Interface
Hardware Resource Interface Layer
Joint Command Interfaces e.g. EffortJointInterface
Joint State Interface e.g. JointStateInterface
Robot Commands e.g. joint efforts - N.m
Robot States e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits Enforce limits (optional)
Effort Transmissions Convert from joint torques to motor torques
(Forward) State Transmissions Real mechanical state
write()
read()
Actuator Efforts e.g. current
Communication Bus e.g. Ethercat, Serial, USB
Mechanism States e.g. encoder ticks
Real Robot
Embedded Controllers e.g. PID loop to follow effort setpoint
Actuators Servos, etc
Encoders Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

Figure 3.6: Overview of the ROS Control framework [50]. It shows all of the necessary components and the data flow.

hardware interfaces need to be defined for each robot model. Every piece of addressable hardware is managed as a *resource*. By default, one resource can only be used by one controller at the same time. This exclusive ownership is managed directly by the corresponding hardware interfaces.

3.7 MoveIt!

MoveIt! is a widespread, open-source collection of software and tools for motion planning, manipulation, 3D perception, kinematics, control and navigation [51]. The framework is based on ROS and utilizes the plugin-based architecture using a centralized node called `move_group`. This node is designed to be light-weight offering high-level capabilities that can be leveraged via different interfaces (python, C++, Rviz Plugin). Most of the functionality such as the IK-resolver is not natively implemented so MoveIt! depends on different packages. In the presented approaches the forward/inverse kinematic service and collision checking feature will be used. [52]

4 State of the art Approaches

When developing an application for a robotic system, the software is usually designed based on the hardware. That means whenever hardware parts will be switched, changes in software have to be made too. As earlier explained in section 3.6, ROS counters this problem by its architecture allowing reuse of code. In this work, these ideals will be carried on and no teleoperation system setup will be developed. Instead, the creation of a universal interface based on ROS is in focus. This means the software is not restricted to specific hardware but it acts as a central node connecting any kind of input device to any kind of robot. This allows quick integration of teleoperations on any type of hardware as long as it can be used with ROS. To further specify this interface, the following criteria have been created:

1. The interface should be universal. As earlier explained, no specific robot or input device should be assumed. This also requires the use of common controllers increasing the number of supported robots.
2. The interface ensures usability. The robot should not move unexpectedly and the delay between the input and the performance should be adequately small. Larger delays would lead to loss of control.
3. The interface has to support robotic arms with at least 6 degrees of freedom.
4. The interface should offer relative jogging as well as absolute jogging. This increases the number of supported input devices.

During research for existing teleoperation solutions for ROS, mainly two approaches seemed to be suitable for this work: The `jog_arm` package from MoveIt! [53] and the `jog_control` package from the *Tokyo Opensource Robotics Kyokai Association* [36]. Both approaches offer a teleoperation interface usable with different robots and input devices. In the following sections, they will be analysed regarding functionality and performance. The better-suited package will then be adapted to meet the described criteria.

4.1 Analysis of jog_control

The package `jog_control` [36] is an open-source ROS package used to jog robotic arms. It is intended to work with all kinds of manipulators and input devices. The package offers two different kinds of teleoperation controls: **Joint jogging** where a selected link can be jogged in a positive or negative direction in joint space and **Cartesian jogging** where the user controls the end-effector in Cartesian space. Only the latter one is interesting for this work which is why the joint jogging will not be further covered.

4.1.1 Integration

In order to teleoperate a robot using the `jog_control` package, the steps published on the GitHub repository [36] need to be followed. At the current state, the package needs to be installed from source. The necessary instructions are on the GitHub repository as well. Furthermore, the `jog_control` package relies on the MoveIt! framework meaning the robot needs to support MoveIt! (see section 3.7).

After installation, a configuration file for the desired robot may need to be created in the directory `/jog_launch/config` in case no matching file already exists. This configuration file contains information about the joints names for joint jogging, the move groups and end-effector frames being jogged one at a time, and in some cases the definition of the `FollowJointTrajectory` controller that is going to be used by MoveIt!.

To use the configuration file, an additional launch file need to exist in the directory `/jog_launch/launch`. With this file, the jogging node called `jog_frame_node` can be started using the previously defined configuration file. Existing launch files offer to optionally run other nodes as well such as the `move_group` node (see section 3.7) or the Rviz program.

4.1.2 JogFrame message as User Input

Sending commands to the `jog_control` teleoperation process is done by publishing ROS messages. The message has to be of type `JogFrameMsg` and needs to be published to the `/jogFrame` topic. A brief overview of this message type is given in Table 4.1.

The message begins with a ROS header property giving details about the chronological order and the `frame_id`. This frame has to be set to the robots base frame because it specifies the root frame for later kinematic calculations. The chronological order is determined by the time stamp. Messages older than the last message received will be ignored. The `group_name` specified the MoveIt! joint group. This can be for example the left arm or the right arm of a humanoid robot. The `link_name` is the end-effector that should be jogged by the user. This link has to be part of the specified joint group. The `linear_delta` describes the 3D position displacement in meters relative to the current position. The `angular_delta` works similar but concerning the orientation. These values are given in axis angles. The relative changes are evaluated in radians and added to the current orientation. Both deltas are directly applied as soon as the message arrives and the robot will try to operate immediately.

Name	Type	Short Description
`header`	`Header`	The header consist of a time stamp and a frame id which determines the reference transformation for kinematics.
`group_name`	`string`	The name for the MoveIt! joint group that should be manipulated.
`link_name`	`string`	Specifies the link that should be operated by its pose. The link must be part of the specified joint group.
`linear_delta`	`geometry_msgs/Vector3`	The relative displacement for the end-effectors position in meters.
`angular_delta`	`geometry_msgs/Vector3`	The relative displacement of the end-effectors orientation in axis angles in radians.
`avoid_collisions`	`bool`	Determines whether collision checking should be used or not.

Table 4.1: The content of the *JogFrame* message with parameter description.

4.1.3 Process structure

The `jog_control` node starts with loading parameters from the configuration file. All of the move groups for MoveIt! and the specified end-effector links will be obtained. The package offers to choose whether messages or action services will be used for joint state publishing. To simplify matters, the use of action services will not be covered in this work. Instead, every available controller of type `FollowJointTrajectory` will be loaded.

The process that does one jogging step is initiated every time a `JogFrameMsg` was received (Figure 4.1, A). If the message is older than the last received message it gets ignored because the chronological order should be maintained to avoid unexpected moving when for example a message is delayed for a longer period.

At first, the end-effectors current pose needs to be determined. This gets done by a forward kinematic service. The resulting pose will then be modified by the messages linear and angular deltas creating the desired pose (Figure 4.1. B). The position deltas can simply be added to the current position. The angular deltas will be transformed into

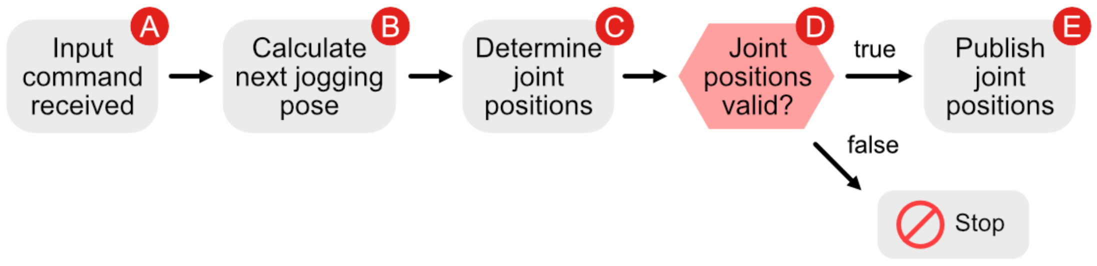
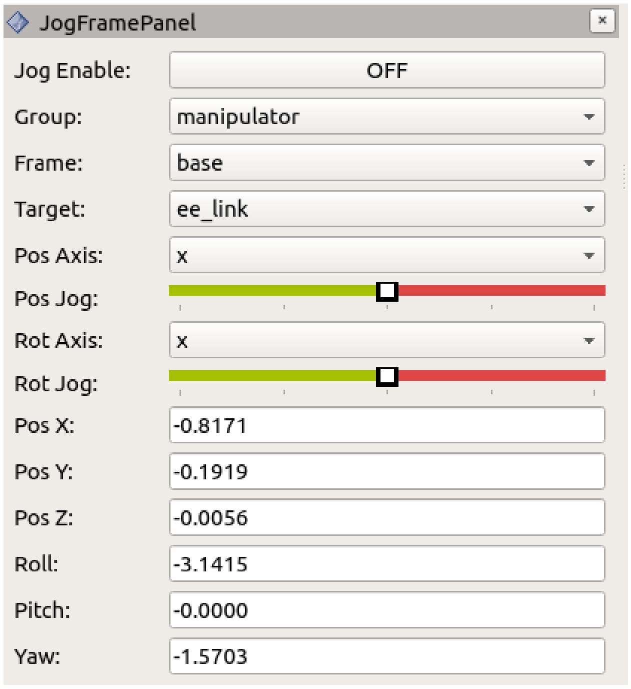


Figure 4.1: An overview of the jog_control teleoperation process. It starts with receiving an input command by the user (A). Based on this command, the end-effectors desired pose will be calculated (B). Then, the joint positions for this pose need to be determined using inverse kinematic (C). If the requested pose is realizable (D) the resulting joint values will be published to a specific controller (E). The process remains stopped and starts over as soon as a new command is received.

a quaternion and then multiplied with the current orientation.

The new position and orientation will be passed as a pose to the inverse kinematic service determining the new joint angles of the specified move group (Figure 4.1, C). Now the solution will be checked whether it is realizable or not (Figure 4.1, D). When no solution was found, the jogging step gets cancelled. Otherwise, the solution will be checked whether too large changes in the joint angles occur. When at least one joint moves more than 90° the process will be also cancelled. This happens due to the handling with joint limits. Joint limits are often at the position -180° and 180° which means the joints position needs to be between those values. When a joint is close to a limit and the jogging command would exceed this limit, the IK service would find a solution near the opposite limit. For example, if one of the joints current position is 175° and the jogging command would require the joint to move by 6° in a positive direction, the IK resolver sets the new position not to 181° but to -179° because this angle is equal to 181° but also inside the joint limits. The joint would now be required to move by 354° in a negative direction which is most likely not foreseeable by the user. Only if zero of those occurrences could be found, the new joint states will be published to the specified controller (Figure 4.1, E).

As earlier explained in section 3.3, singularities may occur in some scenarios and need to be handled properly. When the user jogs the end-effector towards the edge of the manipulator's maximum range two of the links may align and the robotic arm would be singular. In this case, it would not be possible to free the manipulator from its state by using inverse kinematic to jog the end-effector somewhere else. Instead, another program would need to be used. However, the `jog_arm` package would not let the manipulator go into a singular state because the IK-resolver detects the singularity, returning an error.



Jog Enable: OFF
Group: manipulator
Frame: base
Target: ee_link
Pos Axis: x
Pos Jog:
Rot Axis: x
Rot Jog:
Pos X: -0.8171
Pos Y: -0.1919
Pos Z: -0.0056
Roll: -3.1415
Pitch: -0.0000
Yaw: -1.5703

Figure 4.2: The Rviz plugin for the jog_control package.

4.1.4 Rviz Panel

The `jog_control` package comes with a plugin for Rviz, displayed in Figure 4.2. This panel allows the user to jog one specific position axis and one orientation axis. It uses sliders to determine the jogging direction and velocity. The move group, the frame, and the end-effector link can be selected as for example multiple move groups (e.g. left/right arm) can be specified in the configuration file. Furthermore, it displays information about the end-effectors pose.

4.1.5 Evaluation

The integration of the `jog_control` package (see subsection 4.1.1) worked without any issues. The package was successfully tested on a UR5 simulation and a real PR2. The velocity of the actual jogging performance is proportional to the given jogging commands. The package relies on the `JointTrajectoryController` or the use of action services. The collision checking feature is working fine and the overall jogging performance is smooth and responsive enough to operate. The message type is straight forward but it only offers relative jogging. Furthermore, it offers a graphical user interface that may be useful for testing purposes.

Name	Type	Short Description
`header`	`Header`	The header consist of a time stamp and a frame id which determines the reference transformation for kinematics.
`twist`	`Twist`	The twist contains the linear and angular jogging deltas.

Table 4.2: The content of the *TwistStamped* message with parameter description.

4.2 Analysis of MoveIt! jog_arm

The `jog_arm` package [53] is also an open-source ROS package designed to teleoperate robotic arms. It is an experimental part of the MoveIt! framework. In comparison to the `jog_control` package, it also comes with relative Cartesian jogging and joint jogging and is supposed to be hardware unspecific. As in the `jog_control` package, the joint jogging will not be covered in this work. Differences are mainly in the process structure.

4.2.1 Integration

In order to use the `jog_arm` package, the MoveIt! framework needs to be installed from source. The necessary steps are published on the official website [54]. The package can be found in the directory `moveit/moveit_experimental/moveit_jog_arm`. After installation, the package needs to be configured to run on different robots.

A sample configuration file can be found in the directory `/config`. In comparison to the `jog_control` package, only one move group can be specified in the file but overall more parameters can be adjusted.

All the necessary steps to run the package are published on the official GitHub repository [53]. To start teleoperation, the `jog_server` node needs to be started having the previously defined configuration file applied. This can be done with the existing launch files located in the `/launch` directory.

4.2.2 TwistStamped Message as User Input

The `jog_arm` node is controlled using `TwistStamped` messages (Table 4.2). This type is similar to the `JogFrameMsg` (see subsection 4.1.2). Both contain three-dimensional vectors for linear and for angular deltas forming a twist. The difference is that the TwistStamped message does not contain any other fields than the twist itself and a header field. As a consequence, the parameters responsible for picking the end-effector, the move group, and whether collision checking should be used need to be set at the beginning and cannot be adjusted during runtime.

4.2.3 Process structure

As already said, the process structure shows the most differences to the `jog_control` package. Instead of process sequence being triggered every time a command is received, the functional subcomponents are split into three different threads. An overview is illustrated in Figure 4.3.

The node starts with reading all important parameters from the specified configuration file and loading process-relevant resources. From that point on, the jogging thread and, if specified in the configuration, the collision check thread will be launched. Afterwards, relevant topics will be subscribed including the `jog_server/delta_jog_cmds` topic where the TwistStamped messages will be received. Now every thread works asynchronously communicating over shared variables.

The jogging thread obtains the twist coming from the `TwistStamped` message via the shared variables. The current kinematic state of the end-effector is being resolved and at last, the new joint states will be computed. Instead of using services, the inverse kinematic solutions are calculated on a low level using a `RobotState` instance from MoveIt!. After the jogging step is determined, the new joint states are passed to the shared variables. The jogging thread also acts when problematic cases occur namely collisions, singularities and when joints are close to their limits. The thread responses to these scenarios by decreasing the jogging velocity or total halting if the problem is imminent.

The collision check thread asynchronously checks for collisions and reacts if so. The actual checking is delegated to the MoveIt! planning scene monitor. If collisions have been detected, a velocity scale reducing the distance travelled in one step will be set in the shared variables. This scalar is then used by the jogging thread in the following steps.

The main thread is responsible for publishing the new joint states to the controller. Currently the types `JointGroupVelocityController` and `JointGroupPositionController` are supported.

4.2.4 Evaluation

The `jog_arm` package could be integrated without any issues as well. This time, it was only tested in a UR5 simulation. The necessary `TwistStamped` message was first created via the in ROS integrated publishing command `ros_pub` and later with a joystick. The arm was set to a pose, not near a singularity but often the movements failed with an error message `*_joint close to a position limit`. Whenever the arm was able to move and no error was thrown, the movements were slow on the simulation despite increasing the velocity in the `TwistStamped` message. Furthermore, collisions were not always avoided in the simulation even though the parameter has been set to `true`. Comparing to `jog_control`, this package offers more variables that can be adjusted but at the same time, it is more difficult to integrate.

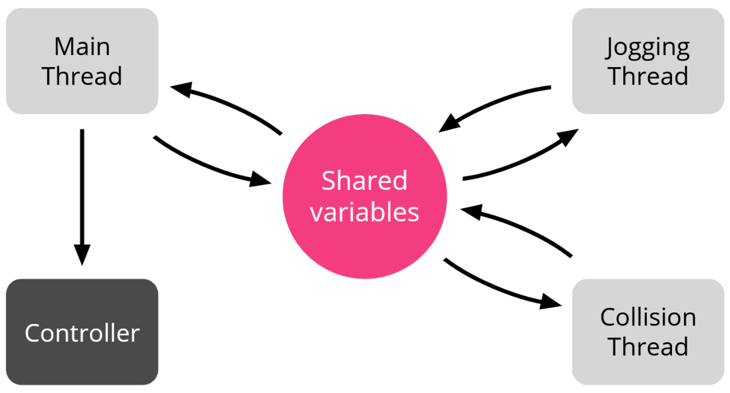


Figure 4.3: A brief overview of the MoveIt! jog_arm process graph. It uses three different threads that communicate via shared variables. The main thread finally sends the joint state data to the specified controller.

4.3 Conclusion

Both existing approaches offer end-effector teleoperation working with relative Cartesian displacement as control mechanisms. While this may be a great solution for joystick-like input devices, it is not suitable for teleoperation systems that have an absolute pose as input such as position tracking systems. Therefore, to increase the number of input devices this feature will be additionally created.

Reviewing the advantages and problems of both packages, at present times the `jog_control` package seemed to be better suited than the `jog_arm` package. The jogging performance is smoother and works as intended in terms of speed and collision checking. Furthermore, the architecture of the `jog_arm` package is more complex in comparison to the one of `jog_control`. But it is also worth mentioning that the development for the `jog_arm` package is very active at the moment so that further comparisons in the future are still of interest.

But not all criteria have been met and so the development on absolute jogging based on the `jog_control` package is in focus. In addition to that, a graphical user interface similar to the in subsection 4.1.4 presented panel will be developed as well.

5 Implementation of Absolute Jogging

One objective for the teleoperation interface is to support a large number of input devices. As earlier explained in section 3.4, there are two different ways to perform jogging: relative and absolute jogging. Both mentioned approaches only offer relative jogging (see chapter 4) so the absolute jogging support needs to be created additionally. During the process of this work, the `jog_control` package seemed to perform better than the `jog_arm` package (see chapter 4) so it is going to be used as the base for this feature. The goal is to create a new node working with absolute jogging commands that can be launched similar to the existing node. Also, a new message type needs to be created because the commands should contain absolute poses instead of relative deltas. To finish it, a similar Rviz panel will be developed allowing the user to experimentally explore the jogging interface.

5.1 Absolute Jogging Message Type

In the original `JogFrameMsg` (subsection 4.1.2) linear and angular deltas in Cartesian space were given to determine the jogging behaviour. These parameters are obsolete for absolute jogging which is why a new type of message was created and named `JogFrameAbs.msg`. In Table 5.1 the contents are listed in detail. The major difference between this message and the `JogFrame.msg` is the pose property indicating the target position and orientation of the end-effector. Additionally, the `damping_factor` was introduced ranging from 0.1 to 1.0. This factor can decrease the velocity when setting it to less than 1.0. A factor of 0.0 would cause the robot to not move at all so this option was excluded. Since the original `jog_control` system relied only on incoming messages, this control system should also be usable by publishing messages to the given topic.

Name	Type	Description
`pose`	`geometry_msgs/Pose`	The pose that the target link should be set to.
`damping_factor`	`float64`	This scalar influences the velocity. It's range goes from 0.1 to 1.0 where the upper limit causes no deceleration.

Table 5.1: The content of the *JogFrameAbs* message with parameter description.

The core functionality is very similar to the original `jog_control` node. But since it does not depend on incoming messages describing relative displacement, it is going to control the robot using a loop. In this loop, the target pose will then be realized stepwise.

5.2 Initialization phase

When the node gets launched, it starts with loading the specified ROS parameters from the configuration file defined in the launcher script. The topic `/joint_states` and `/jog_frame_abs` will be subscribed and additionally, the thread will continue only if a message from the `/joint_state` topic will be received. This ensures proper execution of the loop. For forward and inverse kinematic calculations, respective service clients will be initialized. At last, a publisher will be created for every available controller.

5.3 Updating process

This part of the node is responsible for calculating continuous robot movement. The basic idea is to have a loop continuously do the jogging task while the new target pose gets published asynchronously. A general overview of this loop is depicted in Figure 5.1. The process begins with computing the current pose of the target link (Figure 5.1, A). To do this, the MoveIt! forward kinematics service is leveraged using the robots joint states. The joint state information gets received asynchronously via `/joint_states` subscription.

With the current pose, the position and orientation distances towards the target pose can be determined. The loop will start over early when both distances fall below a certain threshold (Figure 5.1, B). That means the current pose sufficiently matches the target pose. If that is not the case, the next jogging pose will be calculated (Figure 5.1, C). Instead of using the target pose as next jogging pose, the earlier described `damping_factor` can decrease the travelling speed. This gets done with linear and spherical interpolation, where the position and orientation will be calculated separately. The approach will be described based on the position. Let $\vec{d}$ be the difference between the current position $\vec{c}$ and the target position $\vec{t}$ in three-dimensional space (Equation 5.1):

$$\vec{d} = \vec{t} - \vec{c} \tag{5.1}$$

Then, a scaled version of $\vec{d}$ defines the relative displacement for the next jogging position $\vec{j}$. Thus, it needs to be added to the current position $\vec{c}$ (Equation 5.2):

$$\vec{j} = \vec{c} + (\vec{d} \cdot s) \tag{5.2}$$

where the scalar s is the `damping_factor`. The same principle is used to calculate the orientation with spherical interpolation. Both results form the next pose for the target link.

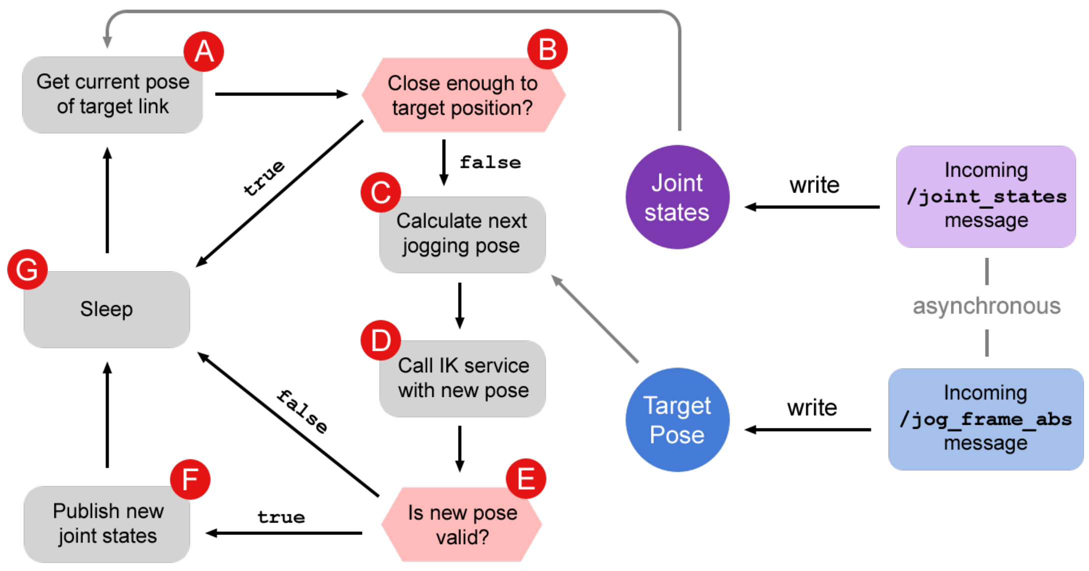

A
Get current pose of target link
B
Close enough to target position?
false
C
Calculate next jogging pose
D
Call IK service with new pose
E
Is new pose valid?
true
F
Publish new joint states
G
Sleep
true
false
Joint states
write
Incoming /joint_states message
asynchronous
Target Pose
write
Incoming /jog_frame_abs message

Figure 5.1: An overview of the core nodes process. The loop (left) gets executed in the main thread while important resources (right) are updated asynchronously.

This pose is now forwarded together with the joint states to the IK resolving service from MoveIt! (Figure 5.1, D). This service offers the option to enable or disable collision checking, hence the `collision_check` property from the message will be passed in as well. The results from the IK service can be put in three different groups:

1. No IK solution was found. This occurs when the given pose is not within the robots range, when collisions were detected or the pose cannot be executed for any other reason.

2. An IK solution was found but at least one of the link's position exceeds its current position by an angle of $\frac{\pi}{2}rad$ $(90°)$. This happens when a joint position limit gets exceeded resulting in an overflow. This prevents the link from doing an entire rotation in the opposite direction.

3. An IK solution was found and all of the links position changes are below the threshold.

The outcome of the IK service determines whether the joint states will be published or not (Figure 5.1, E). If 1. or 2. is the case then the loop will start over early. Only case 3. allows the new joint states to be published to the controller (Figure 5.1, F). The loop then starts over but since it gets executed only ten times per second, the thread will sleep for the respective amount of time (Figure 5.1, G).

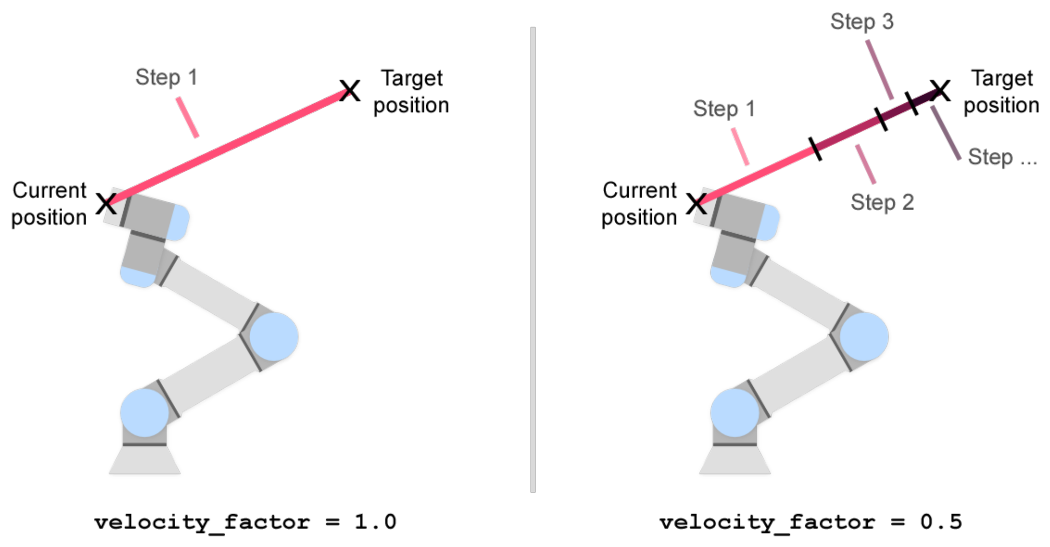


Figure 5.2: The interpolation mechanism demonstrated based on the position. **Left:** The target position can be reached within one step because the damping factor has no effect. **Right:** The first step only involves half of the distance because the damping factor equals 0.5. After this step, the new distance is again multiplied by 0.5 determining the length for step 2 and so on.

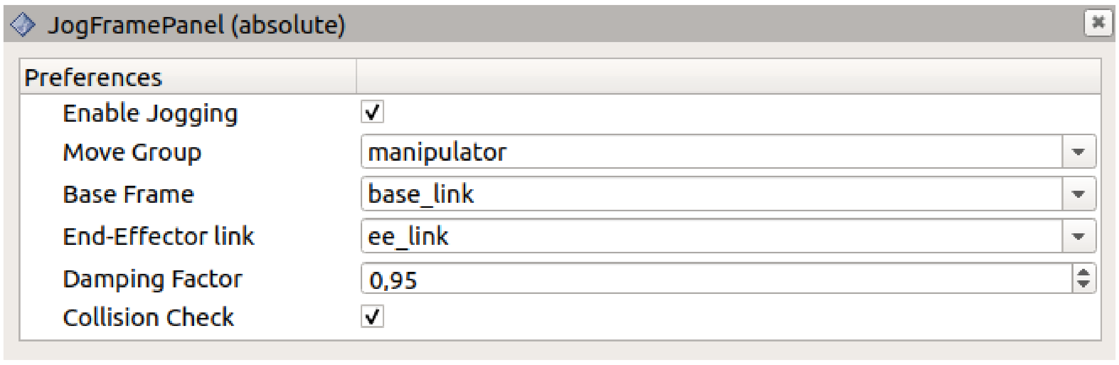
5.4 Rviz Panel

For the absolute control system, a graphical interface could be useful to visualize the target pose. Furthermore, quickly editing the message parameters could help to explore the behaviour of the system relating to its parameters. The goal is to create a Rviz plugin in form of a panel. The plugin should publish the specified messages on the corresponding topic based on its state. The state should be adjustable via UI elements by the user. This panel (shown in Figure 5.3) is oriented on the original Panel of the `JogFramePanel` from the `jog_control` package and uses Qt [55] version 4.8 for more compatibility. It is named `JogFramePanel (absolute)` and consists of the following UI-elements:

Enable Jogging: This checkbox enables or disables message publishing for the panel but not for any other node publishing `JogFrameAbs` messages.

Move Group: This input field determines the name of the MoveIt! move group. Available group names are taken from the configuration file and will be displayed in a drop-down menu.

Base Frame: This input field defines the parent frame of the desired pose. Available frame identifiers will be detected with the TF-library and displayed in a drop-down menu.


JogFramePanel (absolute)

Preferences	
Enable Jogging	✓
Move Group	manipulator
Base Frame	base_link
End-Effector link	ee_link
Damping Factor	0,95
Collision Check	✓

Figure 5.3: The graphical interface plugin for Rviz. The user can enable and disable teleoperations via the Rviz interface.

End-Effector link: This input field specifies the name of the end-effector link. The available links are taken from the configuration file and also displayed in a drop-down menu.

Damping factor: The damping factor can be adjusted with this numerical input field. It is restricted to the defined boundaries (0.1 - 1.0).

Collision Check: This checkbox enables or disabled the collision checking. It is enabled by default but in some cases, it is favourable to turn it off for example when the robot should push or grasp an object.

To intuitively display the target pose, an approach with adjustable x/y/z/pitch/roll/yaw coordinates as input fields seems inappropriate because the user could only edit one parameter at a time. Instead, an interactive marker provided by Rviz is being leveraged (shown in Figure 5.4). The marker can be manipulated in its three-position axes relative to the orientation and three axis angles resulting in a total of 6 degrees of freedom. The orientation will be converted to a quaternion before being published. Without configuration, the interactive marker is not shown. The user needs to add a new display of type `InteractiveMarkers` and then select for `Update Topic` the `/jog_frame_node_abs/update` topic.

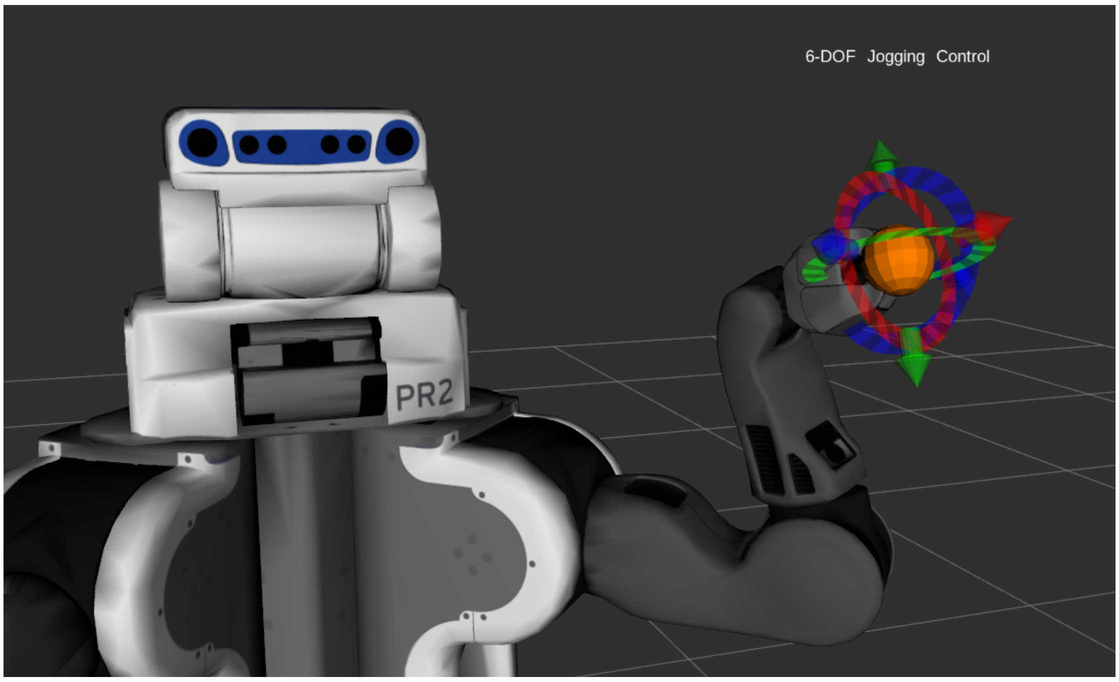


Figure 5.4: The pose of the robot's end-effector can be specified with an interactive marker in Rviz.

6 Evaluation

In this chapter the performance of the `jog_control` together with the in chapter 5 presented extension will be evaluated. The tests will be taken on two different robots: a simulation of the UR5 from Universal Robots and the PR2 from Willow Garage. In section 6.1 the evaluation methods and the setup will be explained while section 6.2 takes a look at the results.

6.1 Evaluation Methods

The performance of teleoperation systems can be evaluated in various ways. A popular way is to let users perform different teleoperation tasks (e.g. pick-and-place problems) [21, 23, 56]. As a result, quantitative data such as success rate, accuracy or time required to finish the task can be collected as well as qualitative data like usability and likeability. For this work, this method cannot be considered since a user study would have been out of the scope.

Other techniques evaluate the teleoperation system using technical aspects of the performance. An analysis of the error between the desired position and the actual position is frequently used as part of the evaluation [20, 57, 58, 59, 25, 60]. A small error indicates that the end-effector moves as requested. But typically this error is related to velocity as faster movements normally result in a larger position error. A more consistent property is the time delay between commanded pose and actual pose [61, 17]. Larger time delays make the robot difficult to control so they should be preferably small.

In this work, the position error and the time delay will be evaluated in different tests. In addition to that, a small course with obstacles has been performed as well. Most of these tests will take place on the UR5 simulation. The input pose will be provided by using AprilTags [62]. AprilTags are fiducial markers allowing to obtain their 6 DOF pose from a two-dimensional image. The image comes from a regular camera pointing towards the user. As the markers are detected, the user can move them around to control the end-effector. If not stated otherwise, collision checking is enabled and the damping factor is set to 1.0. The tests and their meaning are defined as follows:

Straight-line test (simulation): The end effector will only be controlled on one axis at a time to follow a straight path with constant velocity. This way a precise time delay can be determined as well as possible differences among the axes.

Eight-course (simulation): The end-effector will be controlled to follow a line with the shape of an eight around two fixed obstacles (see Figure 6.1).

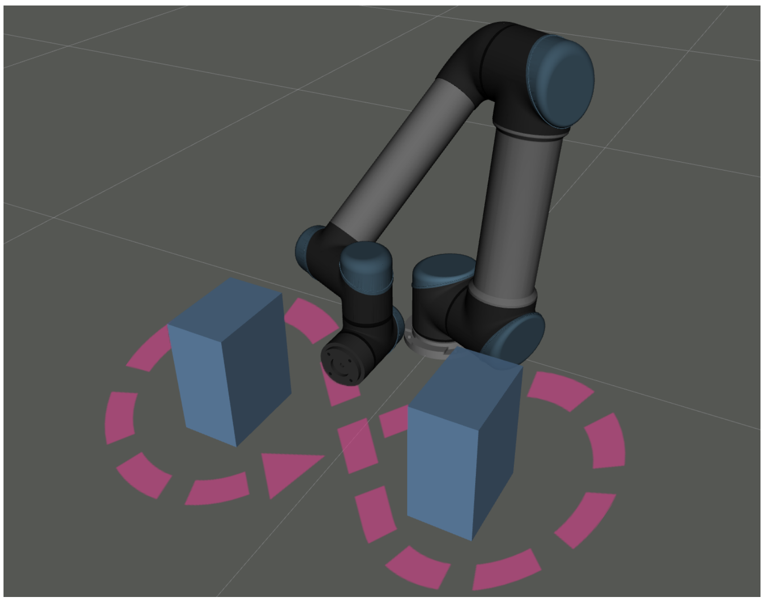


Figure 6.1: The course that the end-effector should follow (magenta line) while avoiding collisions with the obstacles (blue boxes).

Collision test (simulation): The end-effector will be controlled into one of the fixed obstacles and into the robot itself provoking a collision. The robot's behaviour will be analysed.

Arbitrary movements (real robot): The end-effector will be controlled along all axes without following a certain path. This test has the most resemblance to real-world teleoperation tasks so it will be performed on the PR2

6.2 Results

This section shows the results for the previously defined tests.

6.2.1 Straight-Line Test

In this test, the end-effector follows a straight line on one axis at a time. The path is 0.5m long and takes about 5 sec. to complete. The robot receives 20 commands per second generated by a computer program. Figure 6.2 displays the position error of the test performed along the x-axis. It shows that the end-effector moves accordingly with a certain time delay. In the beginning, the error is slightly larger because the end-effector accelerates in that time. The other axes (y and z) remain still as commanded.

The exact time delays are shown in Figure 6.3. On the left side, the graph displays the delay over time. Again, there is a noticeable larger delay at the beginning decreasing

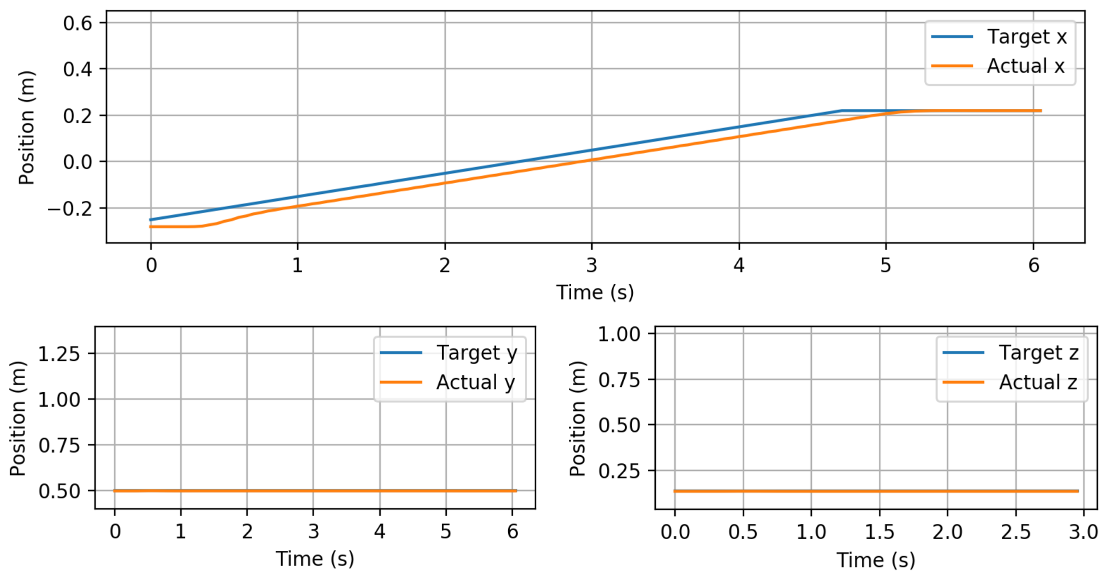


Figure 6.2: The straight-line test along the x-axis. The blue line represents the command while the orange line shows the actual behaviour. The end-effector is commanded to move 0.5m in about 5 sec. along the x-axis. It shows that the end-effector has some delay especially in the beginning where it needs to accelerate. On the y and z-axis, the end-effector remains still. Overall the movement is consistent and smooth.

from 0.5 sec. to 0.35 sec. because of the necessary acceleration. From that point on, the delay remains very consistent at 0.35 sec. until the end where it rapidly drops to 0 sec. since the end-effector has reached the desired position. On the right side, the delays have been summed up for every sent command. The resulting distribution shows that the majority of commands were executed in about 0.35 sec. The results are a bit slower in comparison to the proposed teleoperation system of Park et al. having delay times about 0.2 sec [61]. This was achieved by optimizing the teleoperation process to the used robot. The teleoperation interface of this work should remain unspecific in terms of hardware which is why larger delays need to be taken into account. The test has been repeated for the y-axis and the z-axis as well without showing any significant differences to the previous results.

6.2.2 Eight-Course

The straight-line test is preferable to analyse the position error and the resulting delay but it is not very comparable with real teleoperation tasks. A more representative approach is made with the eight-course (Figure 6.1). The goal is to control the end-effector around two obstacles while collisions should be avoided as much as possible. AprilTags are used as input device (see section 6.1) providing an absolute 3 DOF position. The

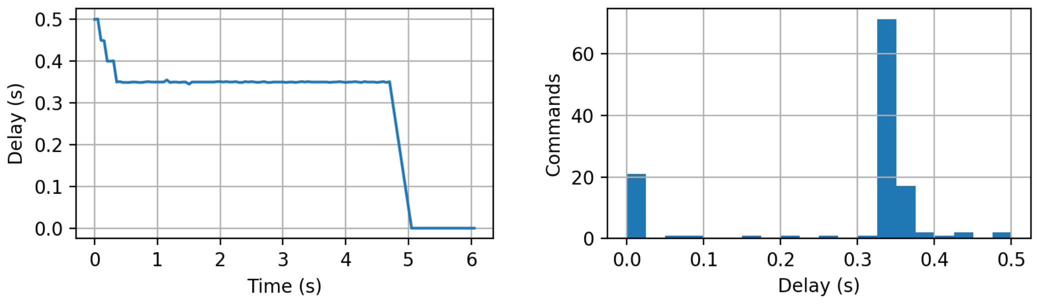


Figure 6.3: The time delay of the straight-line test along the x-axis. **Left:** The graph shows a larger delay at the beginning, then it remains constant until the end-effector reaches the desired position where the delay goes towards zero. **Right:** The chart displays the delay for every command showing again that the major delay time was around 0.35 sec.

orientation control has been disabled because the AprilTags orientation was too noisy and has sometimes led to unexpected behaviour.

The test has been absolved in about 33 sec. without any larger interruptions. Even though the input signal contained some noise, the end-effector moved predictably and smoothly. The positional error is displayed in Figure 6.4 showing that the end-effector followed the commanded trajectory precisely enough to finish the course in a reasonable time. Only after almost 20 sec. one potential collision occurred visible in the top-left diagram. In this case, the end-effector stopped moving on all axes preventing contact to the obstacle.

According to the straight-line test, the delay should be around 0.35 sec. but Figure 6.5 indicates the delay is a bit larger than expected. The left chart shows the distribution of the delays per command. Its maximum is around 0.55 sec. while 95% of the delays are between 0.37 sec. and 0.65 sec. Also, one outlier appears at around 1.7 sec. When taking a look at the right graph, the outlier becomes more apparent showing that it occurs at the same time as the earlier mentioned collision. Overall, the robot was not difficult to control despite the increased delay.

6.2.3 Collision Test

During the eight-test, one collision already occurred but to further analyse the behaviour, multiple collisions will be provoked in this test. Two types need to be considered: self-collisions and collision with the environments.

Self-collisions mean the robots end-effector clashes with a part of itself potentially damaging the hardware. When provoking a self-collision the robot remains still in its current, collision-free position. This position does not automatically assume the end-effector is near part of itself. I.e. it will not jog as close to the desired position as it can but the whole jogging process will immediately come to a halt as soon as the

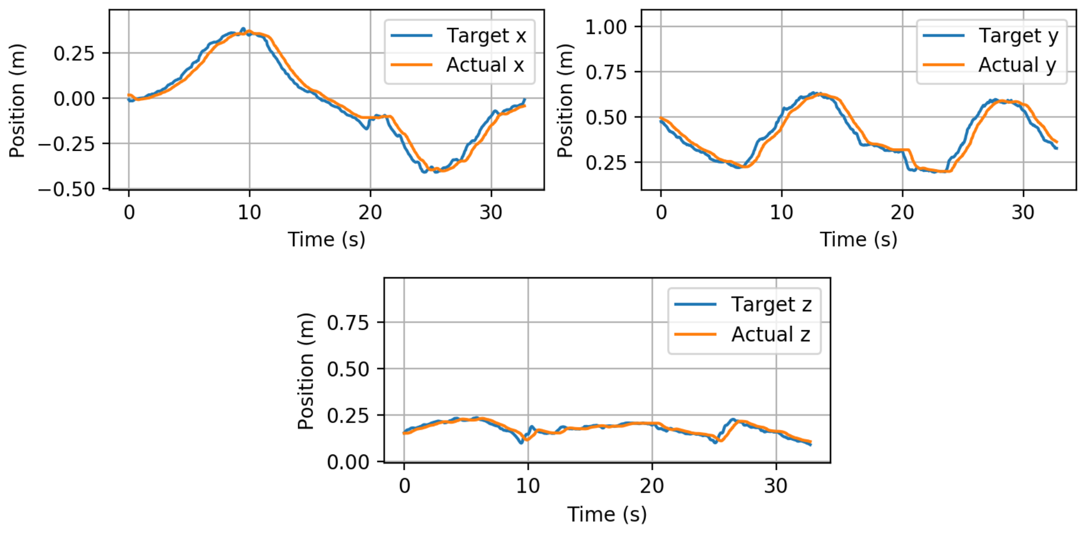


Figure 6.4: The position error of the eight-course. The input signal from the AprilTags (blue) are a bit noisy but the end-effectors movements (orange) remain smooth.

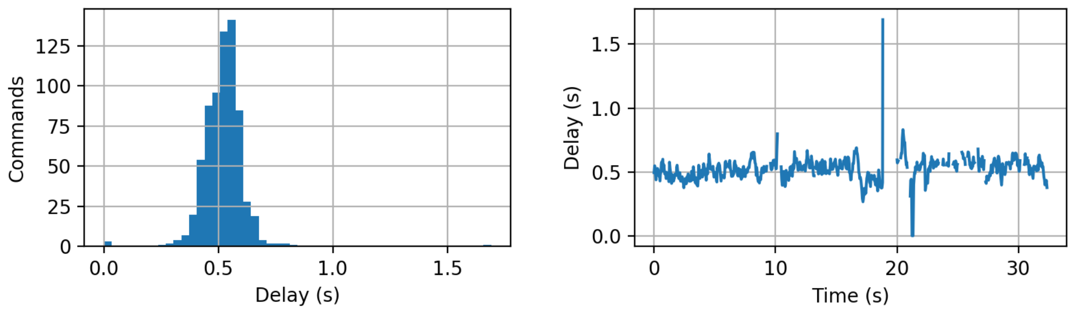


Figure 6.5: The time delay of the eight-course. **Left:** The chart shows the distribution of delay per command. **Right:** The graph depicts the delay over time.

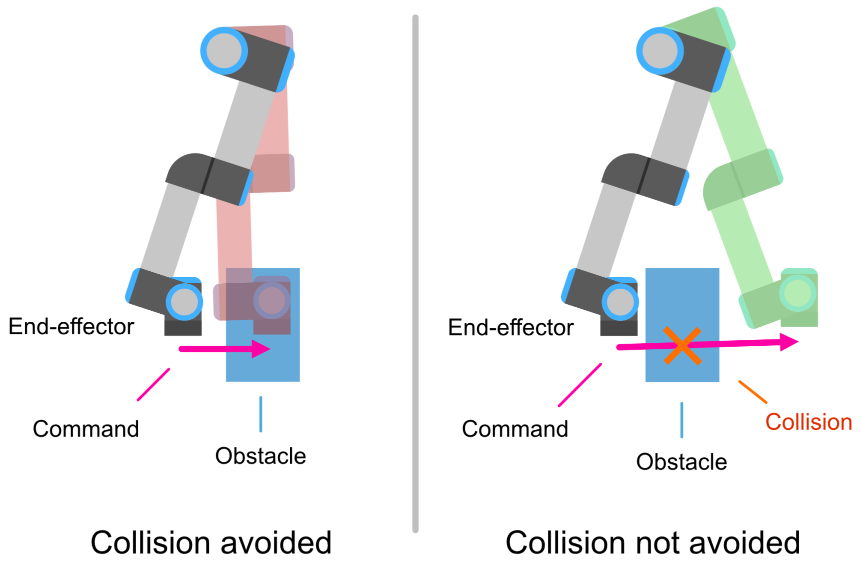


Figure 6.6: Collisions are not always avoided. **Left:** When the command (magenta arrow) makes the new robots state (red) intersect with an obstacle (blue box), the collision will be detected and avoided by stopping the robot's motion. **Right:** When the command results in a state without collisions (green) but an obstacle blocks the motion path, the collision will not be detected and the robot will clash with its environment.

dangerous command was received. This behaviour requires the collision-checking flag in the `JogFrameAbs` message to be set to true.

To check collisions with the environment, the IK service needs to be aware of any obstacles. This means they need to be defined before the teleoperations take place. When provoking a collision with an obstacle (Figure 6.6, left image), the behaviour is similar in comparison to self-collisions. The jogging process stops as soon as a command was received that would require the robot to intersect with its environment. But during the test, another scenario has been observed. When neither the desired position nor the robots current position would cause any collisions but during the motion to the target position, the robot collides with its environment (Figure 6.6, right image). This happens due to missing collision checking for waypoints between the current and the desired position and needs to be taken into account by the user. Similar behaviour for self-collisions could not be produced.

6.2.4 Arbitrary Movements

The final test consists of random movement on a real PR2 by using AprilTags again. The purpose is to observe the teleoperation performance in real-life conditions and compare it to the previously observed behaviour. During the test, there was a noticeable amount

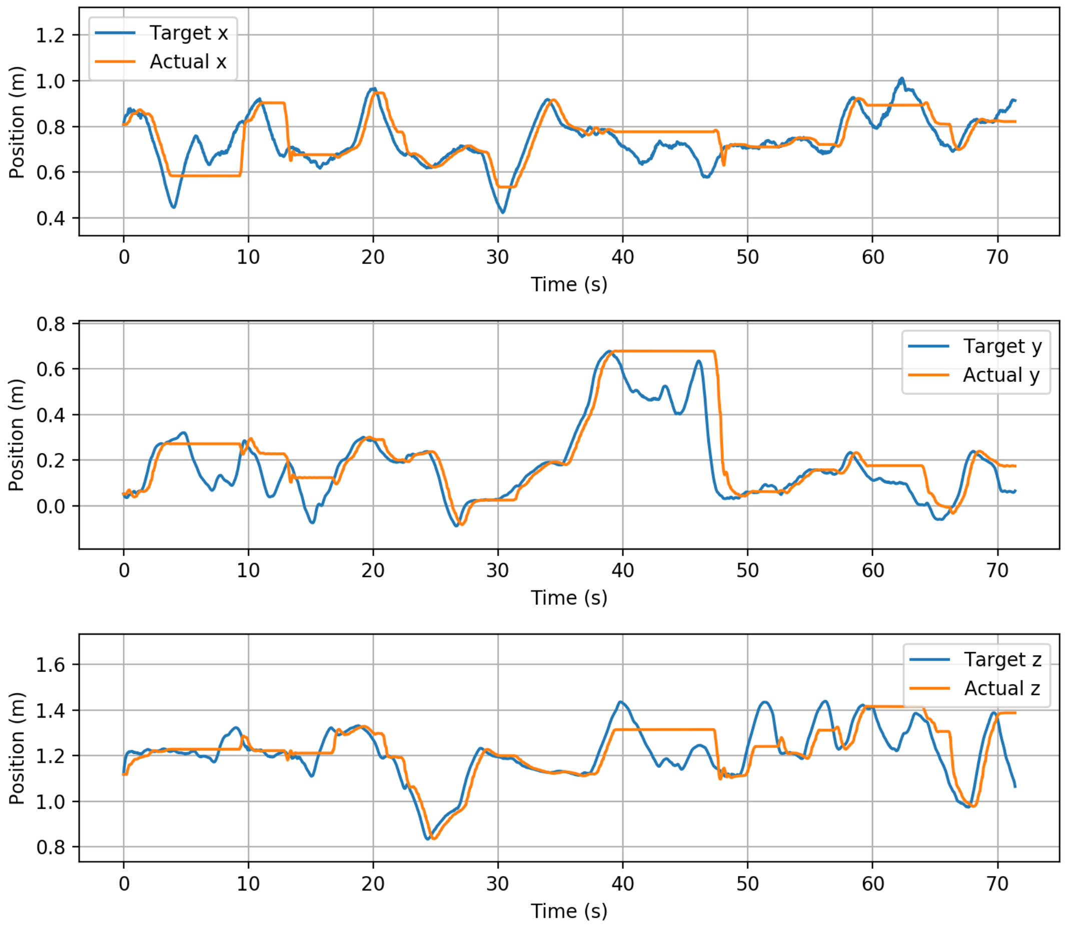


Figure 6.7: Arbitrary movements on a real PR2. Each graph displays one of the position axes. The end-effector (orange) could often not follow up to the input signal (blue).

of movement issues consisting of the robot to stop even though it should have been possible to jog into the respective position. In Figure 6.7 these issues are visible as straight lines that occasionally appear in all position axes simultaneously. These outages made the control difficult even though in absence of these issues the teleoperation behaved as expected.

As a consequence, the delay times are very inconsistent. Figure 6.8 shows missing parts in the graph being the result of the end-effector often not following up to the commands. The remaining delays are mostly located around 0.5 sec. similar to the previous results from the eight-course test. Furthermore, outliers of up to 6 sec. can be found often in relation to the outages.

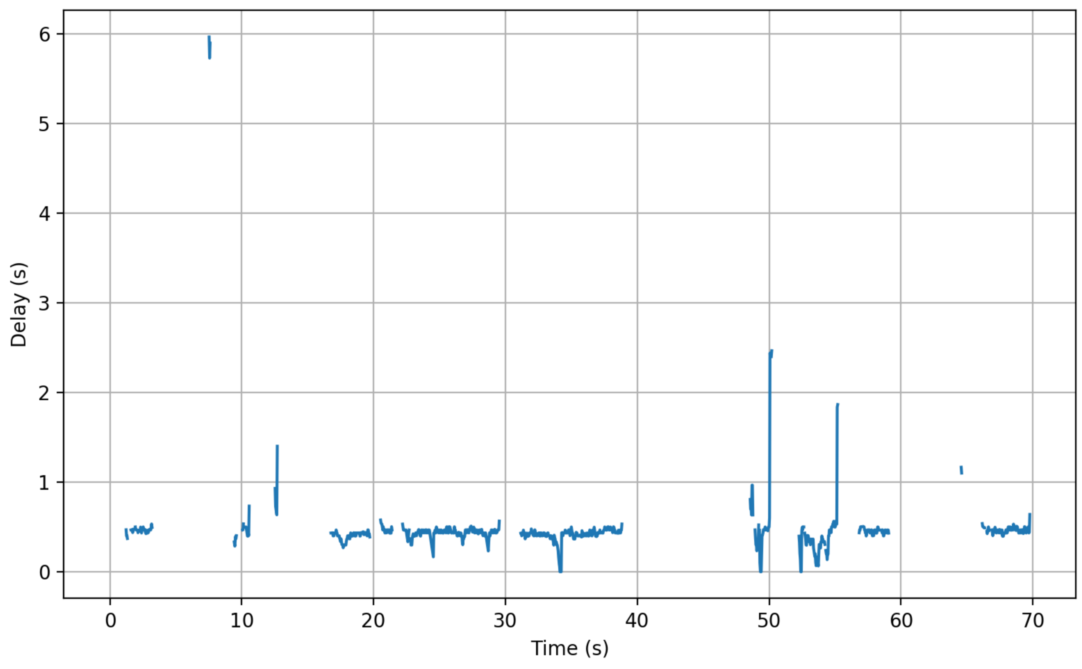


Figure 6.8: The delay of the arbitrary test on the PR2 over time. It is very inconsistent. The missing parts are a consequence of the outages meaning the end-effector could not follow the command at all. The remaining graph is mostly located around 0.5 sec. but also shaped with outliers.

7 Discussion

In this chapter, the evaluation results from chapter 6 will be discussed. Overall, it shows that the teleoperation interface could be integrated and used on two different robots. However, it has some flaws in terms of collision behaviour and robot compatibility.

The straight-line test presents decent delay times on the UR5 simulation under laboratory conditions. The interval of 0.35 sec. was the lowest delay measured during the evaluation. Possible reasons why these values could not be achieved in other tests are for example the simplicity of the trajectory and the consistency of the input signal. A straight line along one axis may not be as difficult as moving the end-effector in a shape of an eight along all three axes. Furthermore, the desired velocity was exceptional steady over time in comparison to human-made input signals. This also may help the robot to react more quickly. Nevertheless, this test shows that giving consistent input signals will result in consistent motion created by the robot.

During the eight-course, command signals were created by humans via AprilTags. This setup represents a more authentic way for evaluation than in the straight-line test. The effects of a noisy and more inconsistent input signal were visible in the measured delay. On average, it increased by 0.2 sec. making the robot slightly less responsive to the input. Despite this, controlling the robot was not perceived as difficult and the course could be finished in reasonable time. Fritsche et al. also stated that delays of up to 0.8 sec. were acceptable to perform teleoperation tasks on their robot [17].

The collision checking feature did not work fully as intended. While self-collisions were avoided in every case, input signals with a too large displacement were not handled properly and collisions with the environment occurred (see section 6.2). This is due to missing collision checks for intermediate waypoints. Only the target state is checked and not the full movement towards it. This behaviour has the potential to be further improved by checking the state along the motion path with a certain frequency. Yet, collisions with the environment will be avoided as long as the commanded displacement is smaller than the dimensions of the obstacles.

The goal of the final test was to observe real-life behaviour on a different robot, in this case, a PR2. Initially, the teleoperation interface worked as intended with measured delays of around 0.5 sec. But at certain times, the robot remained still even though no collisions were imminent and the IK service found a solution. In these cases, the angle of at least one joint would exceed its theoretical limit causing the teleoperation interface to stop the execution for this command. For the PR2, this behaviour can be problematic since it contains two infinite joints in its wrist and elbow [63]. As a consequence, these joints exceed their theoretical limits in rather ordinary poses. As this problem occurred too often on the PR2, safe teleoperation could be ensured since these outages keep the user from performing the task.

8 Conclusion

The goal of this thesis is to have a teleoperation interface for the popular ROS framework that works with different robots and various input devices. The essential components that are used for teleoperation were analysed and explained in chapter 3. Two existing approaches implementing teleoperations for robotic arms in ROS were discovered and analysed in chapter 4. The comparison between the `jog_arm` package from MoveIt! and the `jog_control` package from the Tokyo Opensource Robotics Kyokai Association showed that the latter package was better suited as teleoperation interface. Based upon this, an extension allowing absolute poses as the controlling signal was implemented in chapter 5 and evaluated in chapter 6.

The extension takes ROS messages containing the desired pose and jogs the end-effector directly towards the pose. The velocity can be decreased by giving the damping factor values below one. Furthermore, the user can decide whether collision checking should be enabled or not. In addition to the functionality, a Rviz Panel has been created as well. With this graphical user interface, the absolute jogging can be run from Rviz directly by providing an interactive marker representing the desired pose.

The evaluation showed that the `jog_control` package together with the implemented extension could be integrated into two different robots. On the UR5 simulation the performance is as expected. The delay times are within an acceptable range and the robot moves as desired. On the PR2 the package works very similarly but it has some troubles with the joint limits causing outages. Another aspect is the collision avoidance which fails to work in certain cases.

8.1 Future Work

The teleoperation interface struggles with infinite joints as the software always assumes joint limits and thus will stop when the limits are theoretically exceeded. In this case, the process should include further analysis of the difference between the current joint states and the target states so falsely assumed exceedances can be detected. After detection, the corresponding joints target state could be shifted by full rotations towards the actual joint position preventing the incorrect stopping behaviour of the interface.

Furthermore, the collision-checking feature needs to be improved for safety reasons. As the desired state will be checked for collisions but not the motion path, the robot might clash with its environment when the desired position differs too much from its current position. To prevent this behaviour, the robot's trajectory needs to be checked for collisions at certain times as well.

Bibliography

[1] P. Hämäläinen, J. Takala, and K. L. Saarela, “Global estimates of occupational accidents,” *Safety Science*, vol. 44, no. 2, pp. 137 – 156, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925753505000871

[2] “Glasgow plastics factory explosion – may 2004,” accessed: 16.02.2020. [Online]. Available: http://www.intrescue.info/hub/index.php/glasgow-plastics-factory-explosion-may-2004/

[3] S. Shin, D. Yoon, H. Song, B. Kim, and J. Han, “Communication system of a segmented rescue robot utilizing socket programming and ros,” in *2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, June 2017, pp. 565–569.

[4] “Nuclear decommissioning robotic arm | key-es.co.uk,” accessed: 28.02.2020. [Online]. Available: http://www.key-es.co.uk/?portfolio=nuclear-decommissioning-robotic-arm

[5] “Teleoperated robotic arm for nuclear decommissioning,” accessed: 28.02.2020. [Online]. Available: https://grlab-robotics.com/cas-client/teleoperated-robotic-arm-for-nuclear-decommissioning/?lang=en

[6] Side Zhao and J. Yuh, “Experimental study on advanced underwater robot control,” *IEEE Transactions on Robotics*, vol. 21, no. 4, pp. 695–703, Aug 2005.

[7] B. K. Muirhead, “Mars rovers, past and future,” in *2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*, vol. 1, March 2004, p. 134 Vol.1.

[8] R. H. Taylor, A. Menciassi, G. Fichtinger, P. Fiorini, and P. Dario, *Medical Robotics and Computer-Integrated Surgery*. Cham: Springer International Publishing, 2016, pp. 1657–1684.

[9] S. P. Shoemaker Jr, “Crane game claw gauge,” May 22 2001, uS Patent 6,234,487.

[10] “Defence advanced research project agency,” accessed: 08.12.2019. [Online]. Available: https://www.darpa.mil/

[11] “Darpa robotics challenge (drc) (archived),” accessed: 07.12.2019. [Online]. Available: https://www.darpa.mil/program/darpa-robotics-challenge

[12] “Darpa robotics challenge - finals 2015 (archived),” accessed: 07.12.2019. [Online]. Available: https://haptx.com/robotics/

[13] "Ur5 collaborative robot arm | flexible and lightweight robot arm," accessed: 01.03.2020. [Online]. Available: https://www.universal-robots.com/products/ur5-robot/

[14] V. M. Hung and U. J. Na, "Force control of a new 6-dof haptic interface for a 6-dof serial robot," in *ICCAS 2010*, Oct 2010, pp. 1653–1658.

[15] S. Muszynski, J. Stückler, and S. Behnke, "Adjustable autonomy for mobile teleoperation of personal service robots," in *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*, Sep. 2012, pp. 933–940.

[16] S. Park, Y. Jung, and J. Bae, "A tele-operation interface with a motion capture system and a haptic glove," in *2016 13th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Aug 2016, pp. 544–549.

[17] L. Fritsche, F. Unverzag, J. Peters, and R. Calandra, "First-person tele-operation of a humanoid robot," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, Nov 2015, pp. 997–1002.

[18] K. Goldberg, M. Mascha, S. Gentner, N. Rothenberg, C. Sutter, and J. Wiegley, "Desktop teleoperation via the world wide web," in *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, vol. 1, May 1995, pp. 654–659 vol.1.

[19] L. Ma, J. Yan, J. Zhao, Z. Chen, and H. Cai, "Teleoperation system of internet-based multi-operator multi-mobile-manipulator."

[20] M. Marinho, A. Geraldes, A. Bo, and G. Borges, "Manipulator control based on the dual quaternion framework for intuitive teleoperation using kinect," 10 2012, pp. 319–324.

[21] J. I. Lipton, A. J. Fay, and D. Rus, "Baxter's homunculus: Virtual reality spaces for teleoperation in manufacturing," *IEEE Robotics and Automation Letters*, vol. 3, no. 1, pp. 179–186, Jan 2018.

[22] D. Whitney, E. Rosen, D. Ullman, E. Phillips, and S. Tellex, "Ros reality: A virtual reality framework using consumer-grade hardware for ros-enabled robots," 10 2018, pp. 1–9.

[23] D. Whitney, E. Rosen, E. Phillips, G. Konidaris, and S. Tellex, "Comparing robot grasping teleoperation across desktop and virtual reality with ros reality," 02 2018.

[24] P. K. Artemiadis and K. J. Kyriakopoulos, "Emg-based position and force control of a robot arm: Application to teleoperation and orthosis," in *2007 IEEE/ASME international conference on advanced intelligent mechatronics*, Sep. 2007, pp. 1–6.

[25] M. Çoban and G. Gelen, “Wireless teleoperation of an industrial robot by using myo arm band,” in *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*, Sep. 2018, pp. 1–6.

[26] “Haptx - haptic gloves for vr training, simulation and design,” accessed: 06.12.2019. [Online]. Available: https://haptx.com/robotics/

[27] V. Pruks, K. Lee, and J. Ryu, “Shared teleoperation for nuclear plant robotics using interactive virtual guidance generation and shared autonomy approaches,” in *2018 15th International Conference on Ubiquitous Robots (UR)*, June 2018, pp. 91–95.

[28] J. Burgner-Kahrs, D. C. Rucker, and H. Choset, “Continuum robots for medical applications: A survey,” *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1261–1280, Dec 2015.

[29] J. M. Romano, R. J. Webster, and A. M. Okamura, “Teleoperation of steerable needles,” in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, April 2007, pp. 934–939.

[30] A. Majewicz and A. M. Okamura, “Cartesian and joint space teleoperation for non-holonomic steerable needles,” in *2013 World Haptics Conference (WHC)*, April 2013, pp. 395–400.

[31] A. Tobergte, R. Konietschke, and G. Hirzinger, “Planning and control of a teleoperation system for research in minimally invasive robotic surgery,” 06 2009, pp. 4225 – 4232.

[32] K. J. Waldron and J. Schmiedeler, *Kinematics*. Cham: Springer International Publishing, 2016, pp. 11–36.

[33] I. N. Bronshtein, K. A. Semendyayev, G. Musiol, and H. Mühlig, *Geometry*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 129–268.

[34] P. Corke, *Robot Arm Kinematics*. Cham: Springer International Publishing, 2017, pp. 193–228.

[35] Z. Lai and D. Yang, “A new method for the singularity analysis of simple six-link manipulators,” *The International Journal of Robotics Research*, vol. 5, no. 2, pp. 66–74, 1986. [Online]. Available: https://doi.org/10.1177/027836498600500207

[36] “tork-a/jog_control,” accessed: 27.02.2020. [Online]. Available: https://github.com/tork-a/jog_control

[37] B. A. F. Doyle, John Comstock and A. Tannenbaum, “Feedback control theory.” New York: Macmillan Pub. Co., 1992.

[38] G. Zeng and A. Hemami, “An overview of robot force control,” *Robotica*, vol. 15, pp. 473–482, 09 1997.

[39] “Ros.org - powering the world’s robots,” accessed: 17.02.2020. [Online]. Available: https://www.ros.org/

[40] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[41] V. Tlach, I. Kuric, D. Kumičáková, and A. Rengevič, “Possibilities of a robotic end of arm tooling control within the software platform ros,” *Procedia Engineering*, vol. 192, pp. 875–880, 12 2017.

[42] “Ros introduction,” accessed: 23.11.2019. [Online]. Available: http://wiki.ros.org/ROS/Introduction

[43] “Ros melodic installation instructions,” accessed: 22.11.2019. [Online]. Available: https://wiki.ros.org/melodic/Installation

[44] “Master - ros wiki,” accessed: 25.03.2020. [Online]. Available: https://wiki.ros.org/Master

[45] “Nodes - ros wiki,” accessed: 25.03.2020. [Online]. Available: https://wiki.ros.org/Nodes

[46] “Messages - ros wiki,” accessed: 28.02.2020. [Online]. Available: https://wiki.ros.org/Messages

[47] “Topics - ros wiki,” accessed: 28.02.2020. [Online]. Available: https://wiki.ros.org/Topics

[48] “Services - ros wiki,” accessed: 20.04.2020. [Online]. Available: https://wiki.ros.org/Services

[49] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo, “ros_control: A generic and simple control framework for ros,” *The Journal of Open Source Software*, 2017. [Online]. Available: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf

[50] “ros_control - ros wiki,” accessed: 19.02.2020. [Online]. Available: http://wiki.ros.org/controller_manager

[51] “Moveit motion planning framework,” accessed: 27.02.2020. [Online]. Available: https://moveit.ros.org/

[52] S. Chitta, *MoveIt!: An Introduction*. Cham: Springer International Publishing, 2016, pp. 3–27.

[53] “moveit/moveit_experimental/moveit_jog_arm at master - ros-planning/moveit,” accessed: 27.02.2020. [Online]. Available: https://github.com/ros-planning/moveit/tree/master/moveit_experimental/moveit_jog_arm

[54] “Moveit 1 source build - linux | moveit,” accessed: 29.05.2020. [Online]. Available: https://moveit.ros.org/install/source/

[55] “Qt | cross-platform software developmentfor embedded & desktop,” accessed: 27.02.2020. [Online]. Available: https://www.qt.io/

[56] D. Krupke, J. Zhang, and F. Steinicke, “Virtual Fixtures in VR - Perceptual Overlays for Assisted Teleoperation, Teleprogramming and Learning,” in *ICAT-EGVE 2018 - International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments*, G. Bruder, S. Yoshimoto, and S. Cobb, Eds. The Eurographics Association, 2018.

[57] C. Yang, S. Chang, P. Liang, Z. Li, and C. Su, “Teleoperated robot writing using emg signals,” in *2015 IEEE International Conference on Information and Automation*, 2015, pp. 2264–2269.

[58] G. Du, P. Zhang, J. Mai, and Z. Li, “Markerless kinect-based hand tracking for robot teleoperation,” *International Journal of Advanced Robotic Systems*, vol. 9, p. 1, 07 2012.

[59] Z. Ju, C. Yang, Z. Li, L. Cheng, and H. Ma, “Teleoperation of humanoid baxter robot using haptic feedback,” in *2014 International Conference on Multisensor Fusion and Information Integration for Intelligent Systems (MFI)*, 2014, pp. 1–6.

[60] J. Kofman, Xianghai Wu, T. J. Luu, and S. Verma, “Teleoperation of a robot manipulator using a vision-based human-robot interface,” *IEEE Transactions on Industrial Electronics*, vol. 52, no. 5, pp. 1206–1219, 2005.

[61] S. Park, Y. Jung, and J. Bae, “A tele-operation interface with a motion capture system and a haptic glove,” in *2016 13th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Aug 2016, pp. 544–549.

[62] E. Olson, “Apriltag: A robust and flexible visual fiducial system,” in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 3400–3407.

[63] “Pr2 manual - pr2_manual_r321.pdf,” accessed: 04.06.2020. [Online]. Available: https://www.clearpathrobotics.com/wp-content/uploads/2014/08/pr2_manual_r321.pdf

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Software-System-Entwicklung selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 04.06.2020

Fabian Wieczorek

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 04.06.2020

Fabian Wieczorek