Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Assembly and Programming of a Robot Monkey to Study Imitation Learning in Marmosets

Bachelor's Thesis

Jaú Gretler

gretleja@ethz.ch

Neurotechnology Group
Institute of Neuroinformatics
Dep. of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Joris Gentinetta
Dr. Wolfger von der Behrens
Prof. Dr. Mehmet Fatih Yanik

May 30, 2023

Abstract

The Neurotechnology Group is interested in exploring how marmosets learn from each other. To investigate this process a robot monkey was built, with the intent of letting the robot demonstrate motions to the alive monkeys. The metal body of the robot was designed and fabricated prior to this thesis. This thesis describes the assembly of the robot and discusses how end effector trajectories can be generated from waypoints using the ROS based framework MoveIt.

Contents

Chapter 1

Introduction

Marmoset monkeys are interesting to the global research community, for reasons such as their direct gaze, smooth brain surface and learning by imitation of other group members [1]. In order to investigate their ability for imitation learning it was decided to build a robot, with body proportions and dimensions mirroring the ones of real marmoset monkeys.

In most popular humanoid robot designs the joints are actuated by motors placed at or in close proximity to the joints, since chains, belts and other means of torque transmission introduce slack, friction and an overall increased complexity to the robotic system. However, this placement of the actuators limits the potential for miniaturization of humanoid robots for two reasons. Firstly, the point at which the motors cannot be integrated into the robotic limb without breaking the humanoid appearance is reached quite fast. For example, typical commercially available servo motors have a volume of about $10 - 30\ cm^3$, which results in certain minimal limb diameters. Secondly, there exists a correlation between size of an electric motor and his efficiency, which additionally renders the use of small electric motors for joint actuation less efficient than using motors featuring a higher footprint [2]. As mentioned, an alternative is to place the motors far away from the joints and transmit the torque by e.g threads. Using threads for torque transmission, the torque can be generated by commercially available servo motors outside the robots body and the humanoid appearance can be conserved. This is crucial for the construction and actuation of a robotic monkey measuring no more than 20 cm in height. Thus, the first part of this thesis describes and discusses the process of assembling a robot design based on torque transmission by threads.

In order to program the robot monkey to perform complex movements, a framework is needed which accepts a parameterized pose trajectory and returns a time series of joint states for the robot to follow in order to execute the target movement. There exists a multitude of frameworks for motor and sensor integration or motion generation for robotic platforms [3],[4]. The second part of this thesis describes and discusses how ROS and the MoveIt "motion planning platform" can be used to program end effector trajectories for the robotic marmoset monkey.

Chapter 2

Methods

2.1 Hardware

2.1.1 Metal model

The metal body of the robot was designed prior to this thesis and machined by the **D-ITET Werkstatt** right before the start of this thesis. The proportions and dimensions of the robot closely mirror the ones of marmoset monkeys. Namely, the robot is as tall as it's living counterpart and has the same ratio of arm length to body height.

Figure 2.1: The metal body of the monkey robot

The joint actuation amounts to 5 degrees of freedom (DOF) per arm and 2 DOF for the head.

2.1.2 3D printed parts

The robot is mounted on a 3D printed box, which contains:.

- 16 servo motors
- a Raspberry Pi 4 (called RPP for the remainder of this text)
- a GPIO pin extension board
- up to 14 extension cables connecting the servos to the extension board

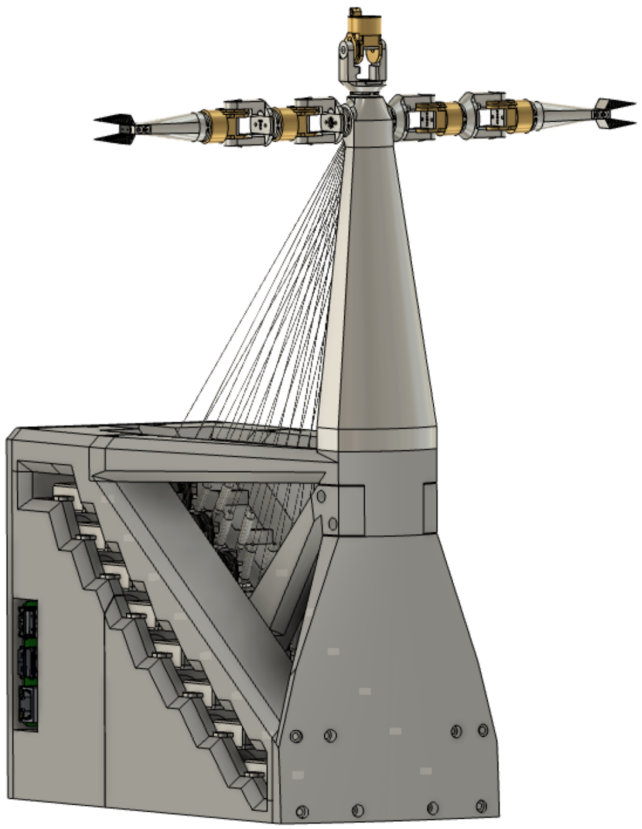


Figure 2.2: The robot mounted on the servo-box

This box, which is called servo-box for the remainder of this text, underwent three design iterations, from which the first two were 3D-printed and assembled.

V1

As can be seen in figure 2.3 the servo-box V1 contains mounting slots for 16 servo motors (8 per side) and a platform for the robot to be screwed onto. In addition, the backplate is removable, such that a RPP and cables connecting it to the servo motors can be fit into a cavity inside the box.

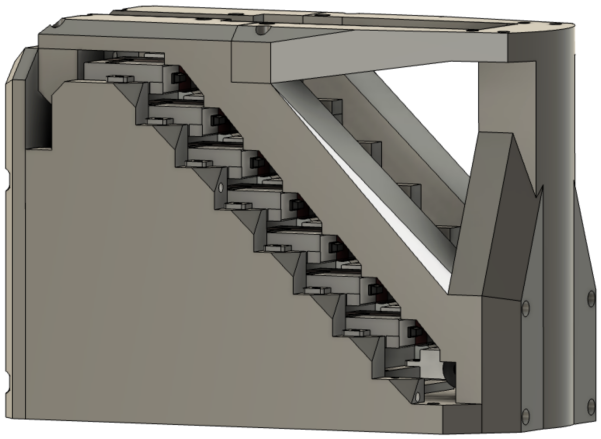


Figure 2.3: Servo box V1

The main problem with V1 was that a wrong servo model was used for its design. As is further described in the motor section, the first servo motor chosen was the KST X10. The 3D model of KST X10 (on the basis of which the screw holes in the base of the box were placed and the base's width determined) was not the correct one, i.e described a different version of the ordered servo. Thus the servo motors could not be be attached to the 3D printed serv-box V1.

V2

To mitigate these problems a second iteration was designed, featuring the following differences:

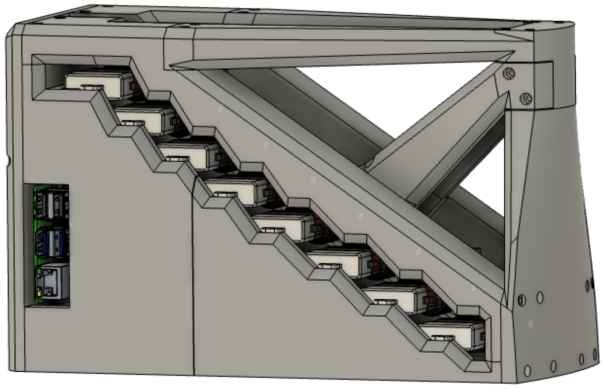


Figure 2.4: Servo box V2

- It is wider, such to accommodate the true height of the servo
- The increased spacing between motors allows for wheels with greater diameter. A greater wheel diameter means more thread length can be pulled which implies a greater ROM for the joint.
- It has two symmetric support pillars at a 45° angle from where the robot stands to where the motors are located. The intention behind these two pillars is to withstand the rotational force exerted from the servos pulling the base of the robot towards the center of the servo box V2.
- The base-part is the bottom part of the box into which the servos are screwed. Upon slicing the STL file of this part in the slicing software it was discovered that it didn't fit into the building room of the 3D-printer. Therefore the part was split in two and the border line modeled in such a way that the two halves could be stuck in to one another (see figure 2.5).
- Two rectangular excisions where made at the far end of the back half of the bottom part. One of them allows some of the output ports of the RPP to be accessed directly and the other one allows air to flow trough the bottom part cavity and cool the RPP's CPU.

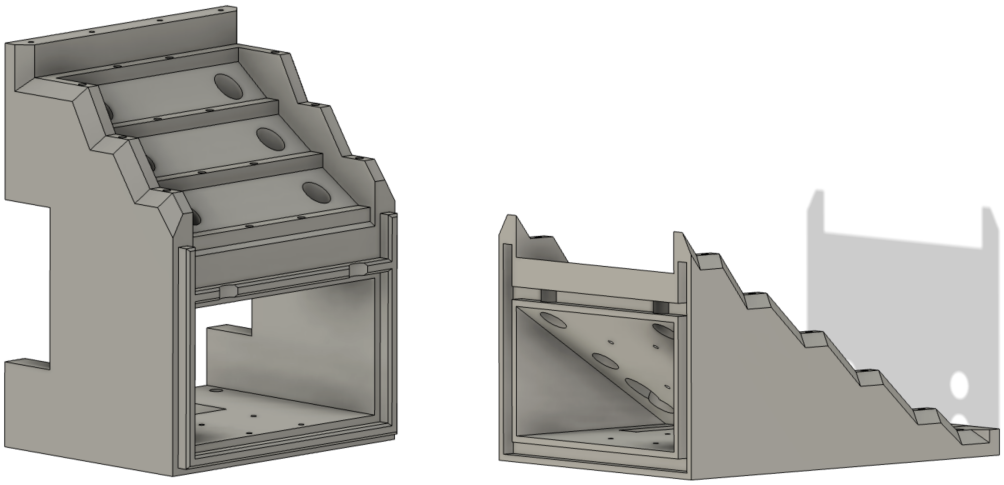


Figure 2.5: The two halves of the base part of V2 can be stuck into one another

The main problem with servo-box V2 was located in the front-part (the part onto which the robot is screwed).

Difficulties arose from screwing in the four M4 screws holding the robot in place. This, because there was not enough space for the head of the screw to be fully screwed into the material and because it was quite hard to access the screws' head with a hexagonal screw driver. It is important for the head of the screws to be fully screwed into the material, since otherwise the 45° support pillars cannot fit their end piece into the roughly rectangular excision left for them in the front part.

V3

The box V3 design looks as follows:

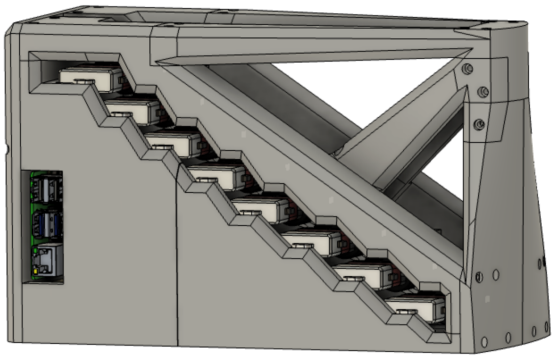


Figure 2.6: Servo box V3

The only new feature of servo-box V3 is that the problems with the front part (described above) were resolved by adding space for the four M4 screw heads and increasing the screw canal diameter to 5mm. The V3 front parts have not been printed yet, since the problems with the front-part V2 could be resolved by manual intervention with a handheld drill.

2.1.3 Power supply

The leading principle behind the power supply circuit was to make it impossible to confuse the 9V supply connection for the motors with the 5V supply connection for the RPP.

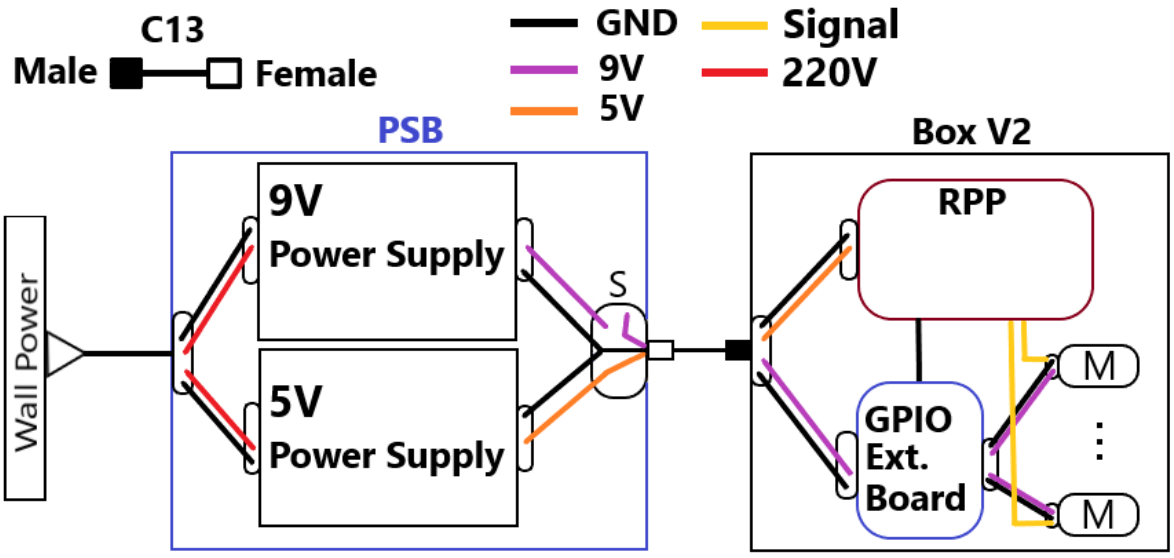


Figure 2.7: The power circuit prevents powering the RPP with 9V

As can be seen in figure 2.7 the power-supply-box (PSB) contains two power supplies, 5V and 9V, for the RPP and the motors respectively. The chosen models are:

- 5V: LRS-150F-5 (Rated current: 22A)
- 9V: PBA-150F-9 (Rated current: 16.7A)

The chosen servo motors (KST A12-T) draw around $3.3A$ at their max torque of 2.0 Nm, adding up to a total max current draw of $16 * 3.3A = 52.8A$. The supply of 16.7A should never the less suffice, since the total max current will not

be drawn for extended periods of time. Current peaks can be mitigated through a capacitor in the pin extension board (which is not installed at the time of writing).

From the PSB a C13 cord carries the common ground, the 5V and the 9V to the monkey robot box.

Since wall power goes directly into the PSB the only removable cable present is the C13 power cord which has a male end at the PSB and a female end at the servo-box V2. Thus it is impossible (by only using the provided equipment) to connect the RPP to 9V.

As an additional security measure a switch was installed at the PSB which switches the 9V line. The idea behind this switch is to be able to turn off the motors quickly in case a faulty program instructs the joints to perform movements which violate joint limits.

2.1.4 Threads

In order to create motion for the joints of the robot, a servo motor is connected to each joint. The connection for a Joint J is formed through two threads, originating from a wheel attached to a servo motor, going through the limbs of the robot and pulling from two opposite sides at J (see figure 2.13).

For the thread a threaded fisherman's line called Spiderwire was chosen. Besides a high tear resistance this fisherman's line also features low friction and no flex.

The threads were installed in the robot according to the following diagrams.

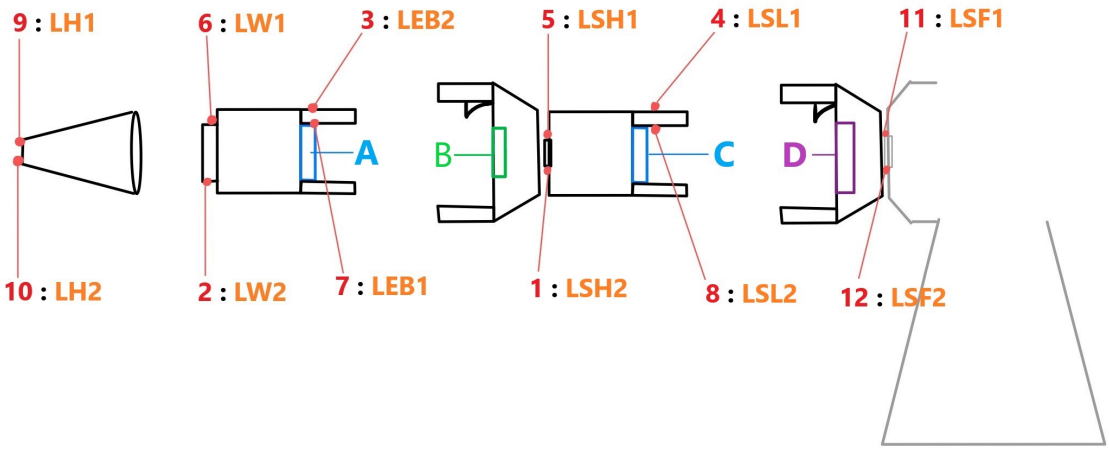


Figure 2.8: Labels for threads (left arm)

The joint IDs used in table 2.8 are described in the table in section F.1.

The following two diagrams show how the threads are guided trough the plates A,B and C which can be seen in figure 2.8.

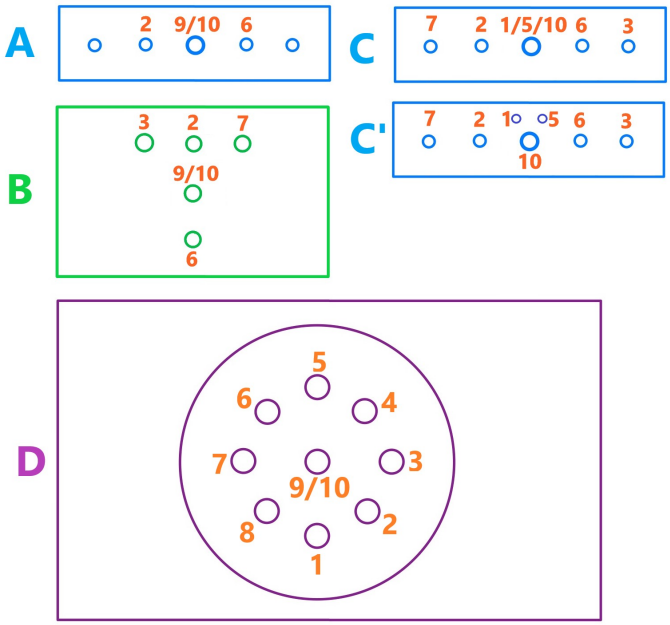


Figure 2.9: Mapping of threads through the plates in the metal arms (left arm), POV: from shoulder. Analogous for the right arm. Section 2.2 discusses the difference C vs C'.

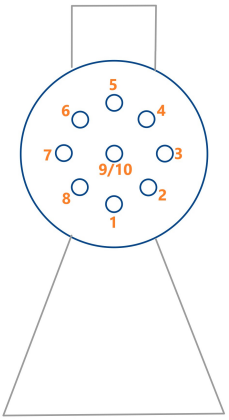


Figure 2.10: Mapping of threads through the shoulder (left arm)

A key aspect of guiding the threads trough the plates and other openings/tunnels is to maintain a symmetry of the threads regarding rotational axes. This ensures that the actuation of one joint, or rather the resulting thread movement, does not affect other joints. There are two different types of joints to consider:

- rotational joints (W,SH,SF): axis of rotation parallel to stretched robot arm
- bending joints (EB, SL): axis of rotation perpendicular to stretched robot arm

For rotational joints, the symmetry is maintained per construction since the threads run parallel to the rotational axis. The main reason why the bending joints are not affected by threads running through them either, is because whenever one thread of a joint is pulled, the other one is automatically loosened. Thus the tension a bending joints experiences from threads running through it remains roughly constant.

2.1.5 Wheels

A central piece of the kinematic chain for the joint movements (see figure 2.13) are the wheels . These are circular 3D printed pieces of resin attached to each servo. All of the wheels were printed using a resin printer. When trying to print the wheels with a FDM-printer it was observed that the inner teeth of the wheel, which lock onto the servo, could not be printed with the desired precision.

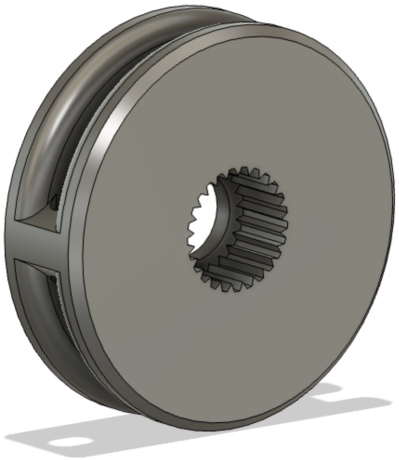


Figure 2.11: Desired shape of inner teeth of wheel

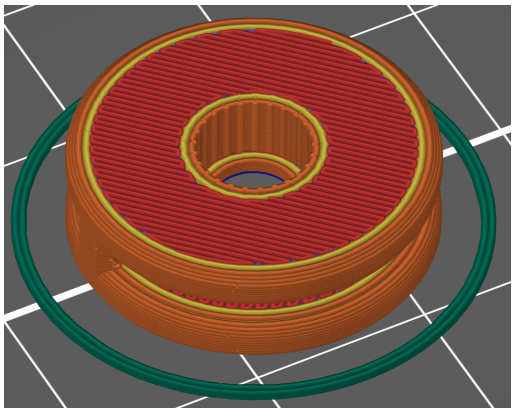


Figure 2.12: Preview of slicer for FDM print: too imprecise

2.1.6 Inconsistent thread attachment

In the servo-box there are two rows of servos, mirrored on the plane spanned by the threads coming from the shoulders of the robot to the servos. Assuming that no two threads leading to a wheel cross each other, the mirroring of the servos implies that the wheels on the two sides of the mentioned plane turn in opposite direction.

This is however not the case, due to assembly issues of the turnbuckles. By the time it was realized that the two screws of the turnbuckles are not interchangeable, some screws had already been attached to threads. Thus, for a small number of wheels, the attached threads had to be swapped.

The mappings in the joint-control-node incorporate these few swapped threads, thus it is only mentioned here as an assembly detail to be considered for future modifications of the robot.

2.1.7 Turnbuckles

The turnbuckles not only provide a single DOF per thread regarding it's tension but also facilitate the assembly process. Their benefit during assembly is that the connection joint-wheel-servo can be severed without actually cutting a thread.

The tension DOF they provide can be used to fine tune each thread such that the rotational range of the servo-wheel can better cover the rotational range of the joints.

However, this DOF brings with it the problem that the turnbuckles can loosen, either slowly over time or quickly if a lot of tension is applied to them. Readjusting can be quite time-consuming.

Thus, after a satisfactory tension configuration had been reached, a screw locking agent was applied to lock the configuration of all turnbuckles. It should however still be possible to manually readjust the turnbuckles.

2.1.8 Joint limits

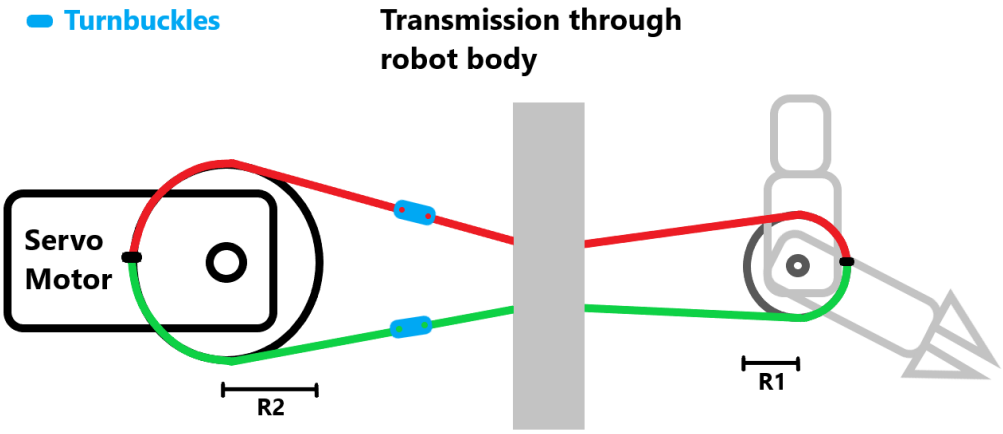


Figure 2.13: The wheels translate the rotation of the servo to rotation of the joints via the threads (The red and green line represent threads)

How far a joint J with threads t_1 and t_2 can be rotated depends on:

1. The mechanical coupling of the two links on either side of J. In the context of this project this is a given, since the links are coupled by ball bearings, with the only limiting factor being collisions of the links among each other (disregarding the threads).
2. The ratio $\frac{R_2}{R_1}$
3. The possible change in thread length δL. δL depends on:
    - The initial length L_0 of the threads. L_0 in turn depends on one hand on the configuration of the turnbuckles and the initial rotation of the wheel
    - The load F_{load} attached to J. F_{load} also includes the friction t_1 and t_2 experience.
    - The thread cross section area
    - The stiffness of the thread

2.2 Plate modifications

As can be seen in the figure 2.9, there are two versions of plate C. Diagram C depicts the state of this plate as it was initially fabricated.

The initial idea had been that threads 1,5,9 and 10 all go through the central hole of plate C. (Note that the last sentence referred to the thread id's introduced in figure 2.8 but also holds true for the analogous case in the right arm).

To manually insert all 4 threads into this central hole of plate C requires fine motor skills but is possible. However, 4 threads in one hole create considerable friction.

In an attempt to reduce this friction one hand thread was removed. This does not hinder the contraction of the hand (needed for object grasping), since the remaining hand thread can be split into two in order to recreate the original symmetric pulling at the hand joint. The release will happen automatically, since the elastic material of the hand will spring into the expanded form as soon as the pulling of the threads stops. Mitigation of friction is further discussed in section 4.1.

2.2.1 Motors

KST X10

As soon as servo-box V1 was printed and the monkey robot was attached it was possible to do preliminary tests with the KST X10 servo motor to determine whether it has enough torque to achieve the desired joint rotations.

The results of these tests were, that all joints could be rotated satisfactorily with the exception of RSH and LSH. The only difference in the kinematic chain between these joints and the others are the lengths of the paths which the threads have to traverse between the wheels and the joints.

Observing that the KST X10 was not able to pull hard enough at the RSH/LSH joints two measures were implemented:

1. Two new holes were drilled, next and symmetric to the central hole of plate C, as represented by the diagram C' in figure 2.9. The effects of this approach are described in section 3.1.
2. New servos with (for our purposes) better specifications were ordered

KST X10 vs. KST AT-12

The following table lists the relevant technical specifications for the KST X10 and KST AT-12 [5],[6]:

Key characteristics		
Characteristic	KST X10	KST AT-12
Torque @8.4V	1.06N	1.8N
Speed/Torque @7.4V	-430 $\frac{\frac{\circ}{s}}{N*m}$	-236 $\frac{\frac{\circ}{s}}{N*m}$
Def. Travel Angle	$\pm 50^\circ = 100^\circ$ Total	$\pm 60^\circ = 120^\circ$ Total

As can be seen in the table, the A12-T:

- has over 70% more torque than the X10
- loses less speed than the X10 when torque is increased

An additional benefit of the higher torque of the A12-T is that it can provide the same torque of the X10 while using a greater wheel diameter. The greater wheel diameter in turn increases the joint's ROM, as described in section 2.1.8.

For these reasons it was expected that the A12-T would perform better than the X10. Both in terms of enabling the robot to rotate the LSH/RSH joints and in terms of smooth operation of the other joints (e.g no noise emissions). Thus there are now 14 servos of type A12-T installed in the robot monkey box, leaving two slots open for two more servo motors to be installed. A possible utilization of these two slots is mentioned in section 4.1.

2.3 ROS

Overview

It was decided to use ROS (the Robotic Operating System) as the basic framework to program the robot. ROS allows for comparatively easy integration and control of sensors and motors of any given robot whose physical characteristics are described in a URDF (Unified Robot Description Format) file

ROS is structured like a graph, with nodes representing different elements of the robot (e.g a sensor node, a motor control node) and the vertices representing communication paths between the nodes.

ROS Nodes

A ROS node is a process that performs computations [7]. One node might compute a trajectory for the robot, another node might run a graphical simulation of the robot.

ROS Topics

Nodes communicate with each other over so called topics which have a certain type. The communication is often organized such that one node *publishes* some information to a certain topic and another node *subscribes* to this topic in order to gain access to the published information

So for example, there is a node called move-group (see section about MoveIt) which, upon successfully planning a trajectory for e.g the arm of a robot, publishes the necessary joint-states to the */controller_joint_states* topic. A second node called joint-state-publisher is subscribed to this topic, reads the joint-states and publishes them in a slightly different form to the topic */joint_states*. A third node, responsible for controlling the motors, is subscribed to the */joint_states* topic and uses the information it reads there to control the motors.

Rviz

Rviz (short for ROS Visualization) is both a simulation tool and an interactive GUI, that can be used to visualize the effects that the ROS network has on a model of the robot.

2.4 Raspberry Pi

One of the useful features of ROS is that the above mentioned nodes can be run on different devices. Thus it makes sense to run computationally expensive code (e.g inverse kinematics solving and path planning) on a desktop PC and only run simple motor control algorithms on a RPP.

The RPP has the additional benefit that it comes with around 40 GPIO (general purpose input output) pins which can be used to transmit the pulse width modulated (PWM) signal generated by the motor control program to the servo motors themselves.

For the above stated reasons a RPP of the 4th generation was used featuring 2GB of RAM.

2.4.1 Pin Extension board

There was a lot of jitter to be observed when the servo motors were first tested. Jitter (the minute oscillation of the servo motors rotational position around a certain angle) can occur when the PWM signal driving the motor is noisy.

A electronic component which can reduce jitter by transforming imperfect PWM signals into cleaner ones is a motor driver board. An additional benefit of such a PCB is that it facilitates separating the voltage supply circuit of the motors and the circuit generating the PWM signal (in our case the RPP).

For these reasons a Pin Extension Board (PEB) was ordered to be stacked on top of the GPIO pins of the RPP.

Since the PEB with a motor output voltage of 9V did not arrive on time a different version of the board was ordered with a motor output voltage of 5V. It was tried to modify the circuit of the PEB such that 9V could be externally applied for the motors without damaging the circuit of the extension board. However, the PEB did not produce any PWM output signal. This was probably due to the manual modifications which must have destroyed some of the circuit.

2.4.2 Reduction of jitter by switching to PiGPIOFactory() for servo control

Initially, the RPP native RPi.GPIO library was used to directly set the duty cycle for each motor [8]. After switching to generating the PWM signals with the PiGPIOFactory() (provided by the gpiozero library), most joints move completely free of jitter. That is, appart from jitter caused by insufficient motor torque to overcome the friction of some thread paths.

2.5 URDF

ROS requires a URDF file of the robot, which must contain all relevant information about the physical parts of the robot. This includes each link's relative position and orientation towards his parent link and for each joint its rotational limits in radians.

2.5.1 URDF exporter

In order to avoid having to create this rather complex file manually it was decided to export the STEP model from the 3D-modeling software SolidWorks (SW) as URDF file. The STEP file for the robot monkey was created during a previous thesis. Instructions for how to export the URDF file can be found in the appendix.

It is worth mentioning that a URDF can but does not have to include collision boxes. When calculating and animating trajectories for parts of the robot, Rviz constantly checks if any of the existing links collide with each other. If no collision boxes are specified, the duration of this computation solely depends on the geometric complexity of the existing robot links. For a typical robot these computations thus are so slow, that Rviz looses it's value as visualization software since the animation is essentially frozen.

By creating simple 3D-shapes, which enclose each link, and including these in the URDF, these computations are fast enough to allow real time collision detection. The collision boxes should have simple geometries and tightly envelop the robots links, as can be seen in figure 2.14.

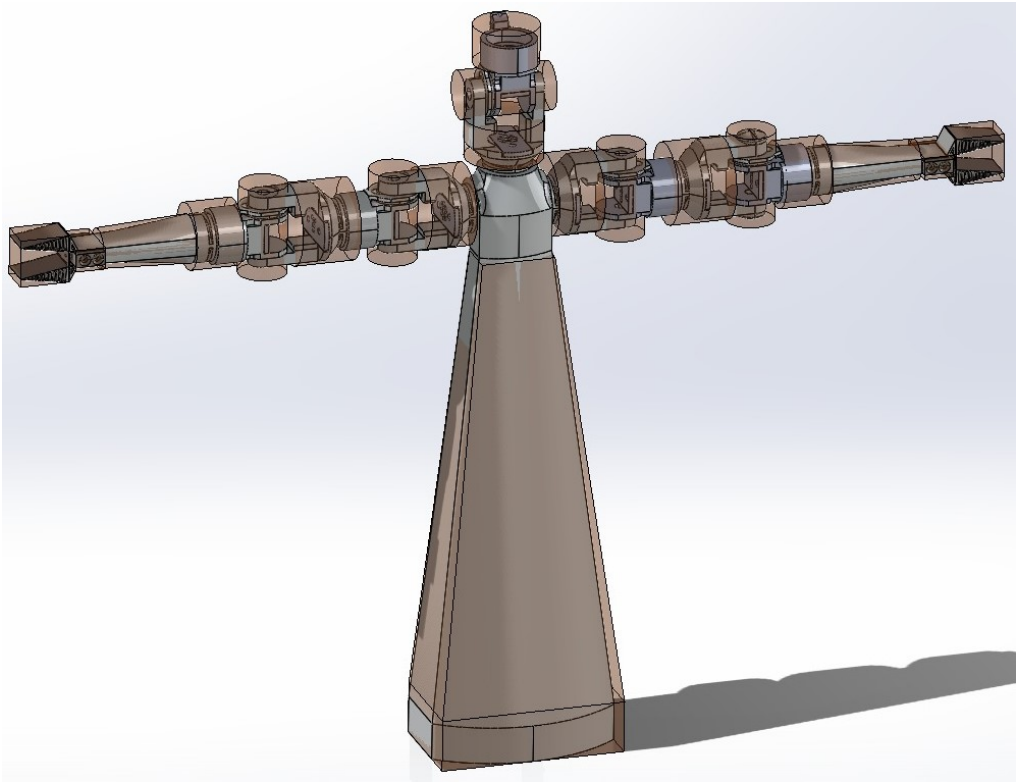


Figure 2.14: The (brown) collision boxes with the (gray) robot underlaid

Note that the collision detection can be further sped up by creating a collision matrix in the MoveIt setup assistant.

2.6 MoveIt

2.6.1 Overview

In order to program movements for the robot it was decided to use the (ROS based) MoveIt "motion planning platform", since it offers features such as:

- Multiple inverse kinematics (IK) solvers including the possibility of implementing and integrating a custom IK solver
- A variety of motion planning algorithms for different applications
- A setup assistant which allows the user to configure MoveIt for a custom robot.
- A tight integration with other ROS related programs such as Rviz and Gazebo, both popular simulation environments for robots controlled with ROS.

It is worth mentioning a few important concepts and introducing some naming conventions.

2.6.2 Planning Groups

Planning groups are composed of a subset of the links and joints of the robot. It makes sense to create one planning group for each kinematic chain of the robot. Following this principle, planning groups for both arms and the head were created using the MoveIt Setup Assistant.

2.6.3 MoveIt Setup Assistant

In order to make use of MoveIt functionalities a so called moveit_config _package has to be created. This can be done by going through different setup steps in the MoveIt Setup Assistant (MSA), the most important being the configuration of the planning_groups. It was decided to create a a different planning_group for each kinematic chain (i.e the two arms seperately and the head). For each planning_group it is possible to set:

- An IK solver (including custom implementations)
- A motion planner
- kinematic parameters such as Goal Position Tolerance, Search Timeout etc.

- Named poses, (e.g "Raised left arm") which contain joint states and can be accessed later through GUI and code interfaces

A detailed description of how the package has to be configured can be found in the appendix in section C.2.2.

2.6.4 Move Group Node

The move-group-node has the structure of a ROS node and serves as an integrator, gathering information about the robot state and providing different interfaces for the user to control the robot [9].

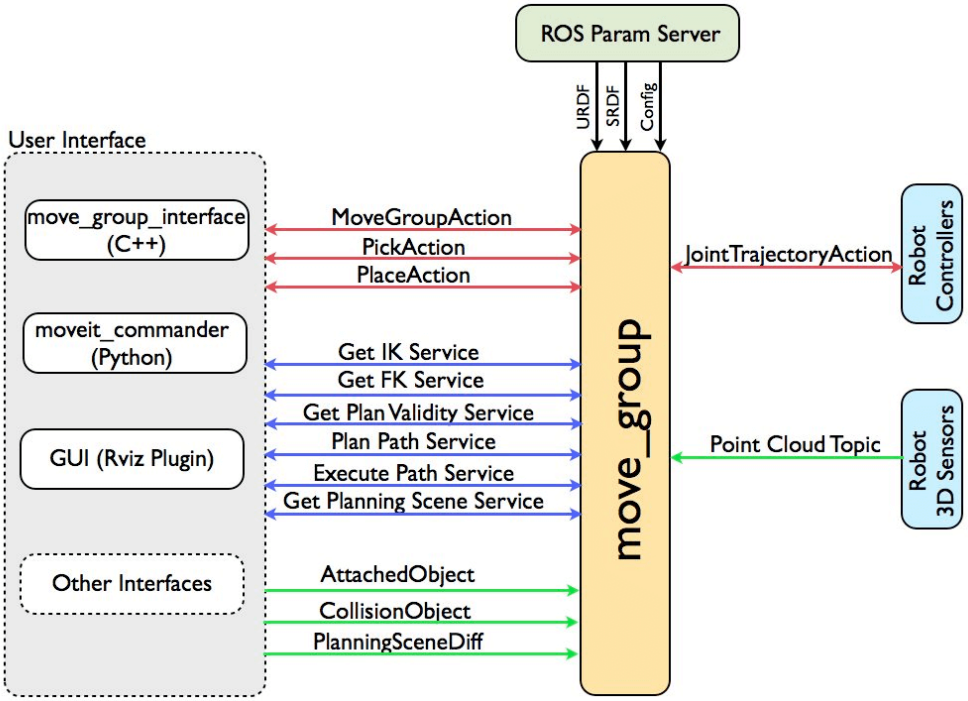


Figure 2.15: High level diagram of the functionalities of the move-group-node (taken from the MoveIt Documentation [9])

2.6.5 Interfacing the move-group-node via the Motion Planning Plugin for Rviz

It is possible to interface the move-group-node via the GUI inside Rviz. If specified so in the launch file (see appendix), the MoveIt's motion planning plugin opens a panel inside Rviz. Through this panel the user can manipulate the joint-states for different planning groups of the robot and plan trajectories for the groups.

This panel offers:

- Visual display of the joint limits set in the URDF. By visually comparing the ROM of the robot in Rviz and the physical robot the accuracy of the joint limits in the URDF can be evaluated (the importance of which becomes apparent in section 3.2).
- The possibility to plan for a single pose or joint goal. By dragging around the end effector (eef) of an arm with a so called interactive marker or by manipulating the joints using sliders (see figure 2.17) we can bring a planning-group into a certain configuration, thus creating a *goal-state*. Then we can plan a trajectory, which moves the planning-group from it's current state (*start-state*) to the previously defined goal state.
- The automatic animation of a trajectory of a planning group from start-state to goal-state.
- The execution of the mentioned trajectory. Note that this will only publish the joint-states contained in the trajectory to the ROS network and that for the robot to physically move, another node needs to use these published joint-states to control the servo motors.
- The possibility of using robot poses defined in the moveit-config-pkg as start- and goal-states. This is convenient because whenever a planning group was brought into some configuration (physically or only in Rviz) and we wish to revert the planning group back to a default pose we defined in the moveit-config-pkg, we can just select this pose as goal-state and then plan and execute the trajectory (see figure 2.16).

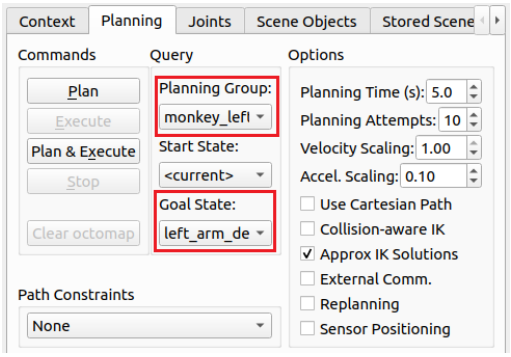


Figure 2.16: MotionPlanning Panel [Planning]

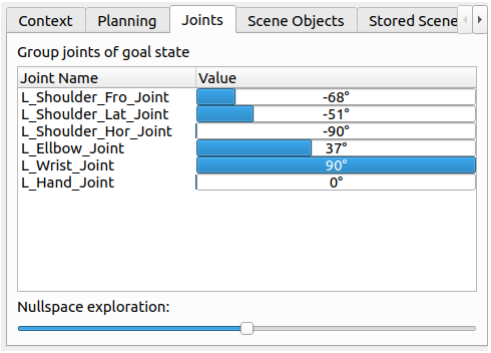


Figure 2.17: MotionPlanning Panel [Joint sliders]

2.6.6 Interfacing the move-group-node via the moveit-commander

In order to programmatically accomplish the actions described in the section above one can make use of the python interface called move-group-commander. The shell based robot control interface created during this thesis consists of different calls to the move-group-commander. In contrast to only planning to a single pose goal, the interface enables the user to construct a trajectory based on waypoints.

2.7 Relaying joint_states via listener node

In order for any trajectory to be executed on the physical robot, the joint-states time series contained in the trajectory has to be read from the ROS network and be used to control the servo motors. To accomplish this a node was written which contains a subscriber subscribed to the *joint_states* topic. Since this node runs on the RPP it can use the received joint-state data to control the robot. This process is further described in section D.4.

Chapter 3

Results

3.1 Friction in the thread paths of joints LSH/RSH

As mentioned in section 2.1.4, each thread experiences a different amount of friction on it's path through the robot. Unsurprisingly, the threads whose paths contain the highest number of turns, experience the greatest resistance. These are the threads connected to the LSH/RSH joint.

During operation of the A12-T motors in the different joints it can be observed that all motors can quietly rotate to min/max duty cycle except the motors attached to joints LSH and RSH.

When the LSH motor tries to rotate to a any given angle, the motor only rotates to a few degrees short of the target angle, stops and begins to beep loudly. If the motor is manually rotated these few degrees such that he rests at the target angle, the noise emissions stop. This behavior will be called range-limit-weakness for the remainder of this text.

The two new holes which were drilled (see section 2.2) and the accompanying reduction of friction for threads 1 and 5 did bring about a noticeable improvement regarding the rotation range of the LSH/RSH joint. Not enough however to eradicate the noise emissions.

In spite of this minimal gain in rotational range it was decided to deactivate the LSH/RSH joints for the remainder of the project, such that only reliable and fully functional joints are included. A straightforward way to deactivate joints it to set their rotational range to 0 *rad* degree in the URDF file (i.e by setting lower-limit and upper-limit to the same value).

Possible approaches to further decrease the friction in joints LSH/RSH are discussed in section 4.1.

3.2 Imprecise joint range data in URDF

A typical ROS setup involves sensors measuring the joint-states and other metrics of the robot constantly and publishing these to the ROS network such that the robot state used for calculations and for visualization is as realistic as possible.

Since there are no sensors used in this project the robot state is solely constructed based on the URDF. There are two problems with that.

First, the whole control of the robot is feed-forward since no feedback is present. Secondly, the calculations and movements are only as precise as the data in the URDF is.

One of the crucial entries in the URDF is the information about the joint limits, which have to be specified for each joint individually in the URDF-exporter.

A problem arises from the fact, that the physical rotational limits are susceptible to different perturbations. For example it happened multiple times during the testing of the motors, that faulty control commands were given and as a result the joints were rotated beyond their usual rotational limits. When this happened, the physical joint limits sometimes shifted a few degrees. Solution approaches are discussed in section 4.2.

3.3 Cartesian path planning

In order to create complex trajectories for the arms of the robot a parametrization of the desired motion is needed. A possible parametrization are points in 3D-space, which, when interpolated, describe a path, that the eef of the robot arm can follow. For the remainder of this text, such points will be called *waypoints* and the path will be called *cartesian path*, the latter taking it's name from a specific MoveIt functionality.

The compute_cartesian_path functionality of MoveIt [10] takes as input a list of 3D points as input and returns a *JointTrajectory* (series of time stamped joint states). This JointTrajectory describes a path for the arms eef from the first to the last waypoint.

As mentioned in section 2.6, it is necessary to implement the move-group-node-interface in order to access functionalities such as the cartesian-path-planner via code.

During this project, a ROS package was written which implements this interface. This package is called Monkey Interface (IM), which is the name under which the rest of this text will refer to the implementation of the move-group-node interface. Having implemented this interface, it is now possible to either hard-code a series of waypoints in the interface itself and use it to compute a cartesian path or to insert the waypoints from somewhere else.

It was decided that an intuitive way of setting the waypoints is to drag the eeff of the targeted robot arm to desired positions in the Rviz GUI and then save this eef position as waypoint.

To achieve this in code, a subscriber was inserted into the MI which listens to the interactive_marker/feedback topic, through which Rviz publishes information (such as pose) about the interactive marker to the ROS network.

Thus it is now possible to manually specify an eef-path of arbitrary length in the Rviz-GUI (see figure 3.1), to compute a *JointTrajectory* for these waypoints and to execute the trajectory on the physical robot (via the listener-node).

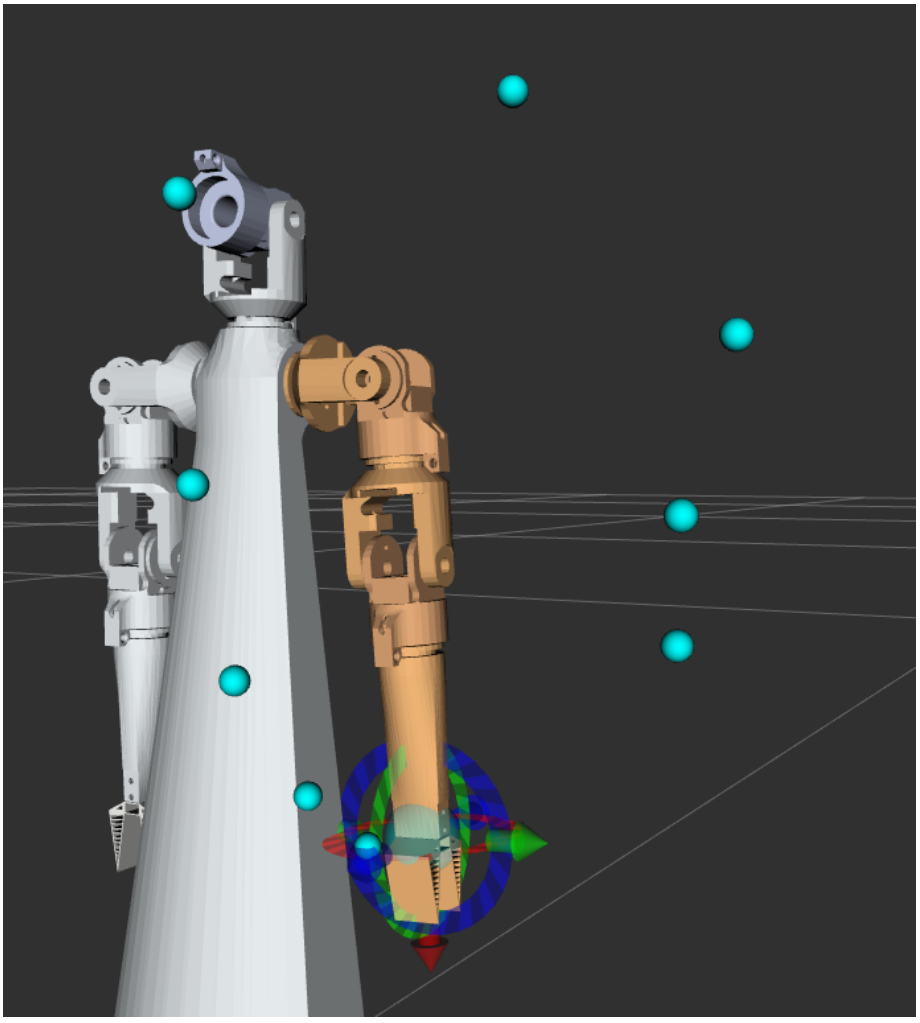


Figure 3.1: The blue spheres describe waypoints which were set manually.

It was found that in order for the compute_cartesian_path method to succeed, its input argument *eef_step_size* had to be set to a value greater than the largest distance between any two consecutive waypoints of that path. This eef-step-size is usually set to a value much smaller than the distances between waypoints, since it refers to the interpolation step size the eef ought to make between to waypoints. If the eef-step-size is set to smaller than the mentioned distance, no valid path is found. During this project no explanation was found as to why no interpolation can take place. However, this has proved to be irrelevant concerning the planning and execution of cartesian paths, since the cartesian paths are made up of straight line segments and the amount of interpolation points on these segments only changes the speed, with which the eef completes the motion. The speed can be adjusted separately.

3.4 Interactive Marker position vs. eef position

3.4.1 Overview

As described in section 3.3 the first step towards creating a cartesian plan involves manipulation of the interactive markers (IM). When an IM in Rviz is used, it's new location and other information are published to the interactive_marker/feedback topic. When developing this method it was first believed that the IM pose being published at time t_0 must be identical to the pose of the eef at time t_0.

The planning attempts made by waypoints, which were set following this assumption, failed about half of the time. It was then understood, that when dragging the IM around in Rviz it was not the final position of the IM which is published (and which coincides with the eef's position) but instead the position in Rviz simulation space at which the IM has last been dragged to by the mouse.

Put differently: The IM can for the sake of the argument be understood as a rope attached to the eef on one side and to the users mouse cursor on the other side. This rope can be used to drag the eef around the Rviz simulation space. When the mouse (and thereby the IM) is released, the last position update sent to the interactive_marker/feedback topic is the position at which the user last held the rope.

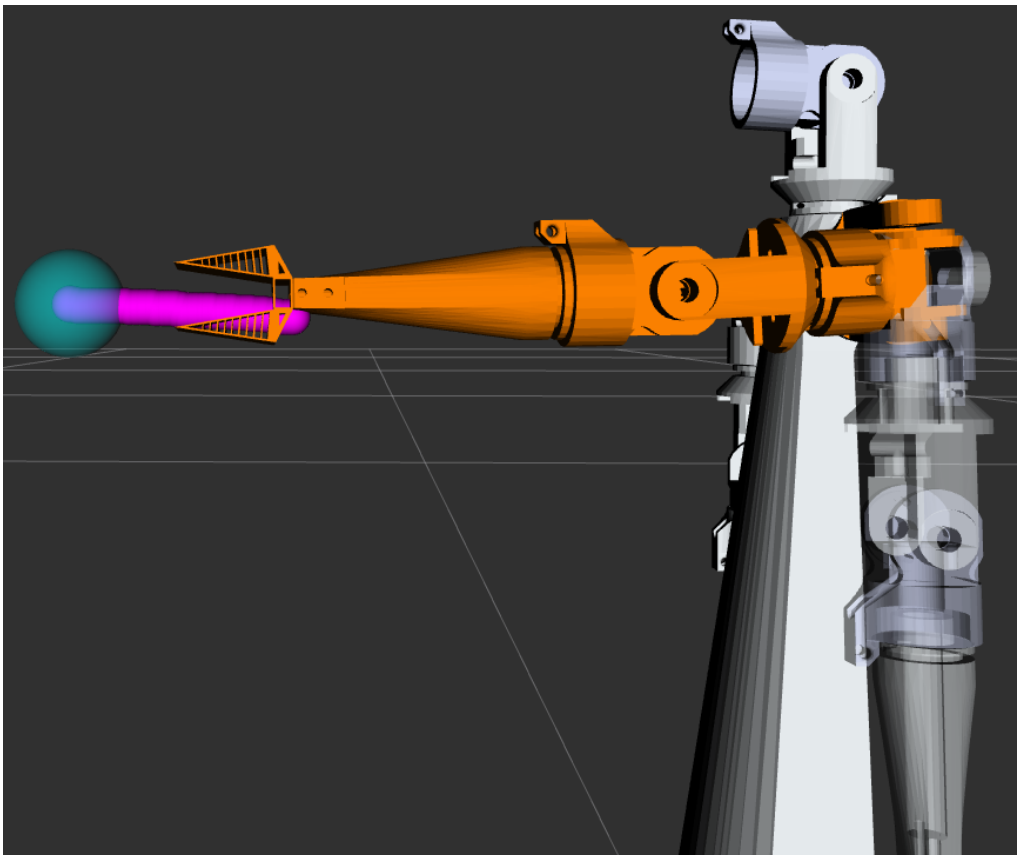


Figure 3.2: The pink spheres display the last few recorded IM positions. The eef of the arm is constantly being dragged towards the IM.

A blue sphere marks constantly the position of the IM, and the rope-effect is visible through the fact that the blue sphere sometimes is not at the same location as the eef, when it is being dragged (see figure

3.4.2 Workaround

To ensure that the last position published really is the one of the eef, the following workaround is suggested. Whenever a waypoint was set (by dragging around one end of the rope) and the IM has bounced back, the user can perform a manipulation without consequences on the IM. The only possible futile manipulation is to rotate the IM, which has no consequences since the robots arms don't have enough DOFs to allow arbitrary orientations. Thus, the last IM postion published is the one where the eef is resting at after the futile rotation.

3.4.3 Automatic IK validation for newly set waypoints

The above mentioned workaround of performing a futile manipulation on the IM itself does not prevent the user from setting waypoints which are outside the robots reach. To mitigate this, a method in the MI was implemented which tries to plan a trajectory from the last eef pose to a newly set waypoint W_{new}. The existence of such a trajectory implies that W_{new} is in the robots reach and that W_{new} can safely be appended to the valid waypoints so far collected. Using this validation method, a user can feel more confident when creating longer waypoint collections. If the user tries to set an invalid waypoint, a error message appears and the waypoint does not appear in the GUI. The check is done in less 200 ms, thus it doesn't add a noticable overhead in the waypoint collection.

3.5 Saving and loading trajectories via the MMGN

To facilitate creation and management of single arm trajectories the following services were implemented in the MMGN:

- Saving of waypoints into a json file (with file name prompt)
- Loading saved waypoints from a json file spcecified by file name. The trajectory thus loaded is displayed and can be executed on any planning group. This is a yet unsolved problem, since a loaded trajectory can only meaningfully be executed on the planning group for which it was created in the first place. It should however be straight forward to include the planning group information in the saved file and check planning group compatibility when loading trajectories.
- Editing a loaded waypoint trajectory and saving the edited trajectory under a custom file name

In addition, the user can also perform the following actions by giving commands to the MI in the shell:

- Plan a trajectory to a hard coded pose goal and execute the trajectory
- Plan and execute a cartesian path from hard coded waypoints

All of the commands in this section can be issued for both arms. The user is prompted for the target planning-group by the MI. For the head planning group currently only joint-state goals can be executed.

Chapter 4

Discussion

4.1 Solution approaches concerning excessive friction in LSH/RSH joints

The problem named range-limit-weakness mentioned in section 3.1 also occurs if the target angle is the min or max duty cycle. This implies that the A12-T lacks only very little torque, to be able to overcome the above average friction of the LSH/RSH joint

Since the A12-T works perfectly for all other joints, it is doubtful if acquiring motors with a stronger torque is reasonable. The costs have to be compared to the efforts needed to decrease the friction of the LSH/RSH threads. Possible solution approaches for that might be:

- Decrease friction of the edges of the holes in the plates through which the problematic threads traverse. It is probably worth refabricating plate C', since the two newly drilled holes could not be post-processed in the same way the other holes had. Namely, the circular edge between hole and plate is sharper than it is for the other holes.
- Decrease wheel diameter of wheels on motors connected to LSH/RSH, which would increase the torque applied to the threads.These two joints only require a 90° ROM, which wheels with smaller diameter should be able to provide.
- Explore different lubricants. The one applied during this thesis ("Toolcraft: Silikonspray") did decrease the friction noticeably but not satisfactorily.
- Since two servos are not used, it could be tried to attach their threads to the RSH/LSH threads such that two motors pull per problematic joint instead of one.
- Explore different thread materials, find one with less friction than the one used currently.

- Find a new route for the threads connecting the joints and their respective wheels. This approach is probably very costly since it would mean that some parts of the robot would have to be redesigned and refabricated.

4.2 Solution approach concerning imprecise URDF data

As mentioned in section 3.2 for some joints the physical joint limits diverge from what is written in the URDF. The problem is that this divergence manifests itself in Rviz. It can be visually seen, that the robot state in ROS slightly differs from the physical robot.

The only obvious solution is to readjust the joint limits in the URDF such that they better match the physical limits. This solution would however depend on the configuration of the turnbuckles, wheels and threads to remain constant after the URDF modification.

4.3 Shortcomings of current control mechanisms

4.3.1 Dual arms

Although the MMGN currently allows for trajectory generation for individual planning groups, no method has been implemented yet to control multiple planning groups simultaneously. The following approaches could mitigate this.

LAAS Pick and Place MoveIt Plugin

The "Laboratory for Analysis and Architecture of Systems" (LAAS) based in Toulouse, France has created a MoveIt plugin which enables a MoveIt configured robot to perform dual arm pick and place actions.

Their plugin uses two action servers which together can process a dual arm pick and dual arm place request and generate a planning request [11].

This plugin could potentially enable the robot monkey to pick and place objects within his range. Further, from what can be read in their documentation, it should also be possible to create dual arm trajectories without picking and placing an object [12]. Dual arm trajectories of arbitrary length could potentially also be created, by chaining saved dual arm start-to-end-state trajectories together.

Even though there exists a tutorial on how to use this plugin [13], the complete source code is unavailable to the public. Still, the tutorial should at least get us started on using their plugin for dual arm pick and place actions.

Dual Arm Single Pose Goal

The MoveIt tutorial collection contains tutorials for an example robot called PR2, which also features two arms. The source code for this PR2 robot tutorial contains a section about dual arm pose goals [14]. The approach starts by creating a planning group containing two robot arms (presumably using the MSA). For this new dual_arm planning group, two different pose goals are set, each with a different target eef and a dual arm trajectory is planned for the two pose goals. This method could enable the monkey robot to execute dual arm pose goals.

However, there are two potential implementation pitfalls to consider:

- The planning groups defined in the MSA in this project each contain one kinematic chain, e.g base_link to LH_Link. Since two arms cannot be included in the same kinematic chain and there is no apparent way to include multiple kinematic chains in a planning-group, it remains unclear how a dual arm planning_group can be created with the MSA.
- The documentation for the PR2 tutorials state the code was developed for ROS Indigo, which is 5 ROS versions older than ROS Noetic, the version used for this project. Thus it is unclear, how well the mentioned methods can be adapted for ROS Noetic.

Merging of JointTrajectories

Planning a cartesian path containing multiple planning groups simultaneously is currently not possible with MoveIt. However, it might be possible to create seperate cartesian paths for multiple planning groups and merge the generated JointTrajectories into one JointTrajectory [15]. The main challenges here would be the implementatoin of the JointTrajectory-merging and handling potential collisions of the two eefs.

Multiple move-group nodes

Currently there is one move-group interface being run simultaneously with one joint-state-listener-node (a node subscribed to the joint_state topic). It could be tried to duplicate both nodes, modify the listener nodes such that they each only listen to their respective move-group node and such that they only listen to/relay target joint states for their planning group. Running multiple move-group nodes is possible, as was confirmed during the project.

4.3.2 Synchronization of plans for different planning groups

The two last mentioned approaches for dual arm trajectories, Merging of JointTrajectories and Multiple move-group nodes, could both be used to to synchronize the execution of JointTrajectories of different planning groups.

To synchronize the JointTrajectories by merging, the merging method must allow a temporal offset to be set between the execution of the different JointTrajectories.

To synchronize the JointTrajectories by duplicating the move-group and listener nodes, another node would have to be implemented which contains a JointTrajectory execution schedule. This node would then inform the move-group nodes when they should start publishing their trajectories to their respective listener nodes.

4.3.3 Integration of different saved motions into multi motion sequences

The previous sections discussed how the execution of different JointTrajectories could be implemented and synchronized. In order to store created JointTrajectories and manage them efficiently the same method can be used which is used in the project to save the JointTrajectories describing cartesian plans. This method is the ROS package called rospy_message_converter [16].

This method enables us to save any ROS message (including JointTrajectories) to a json file and to later retrieve the the ROS message from the json file.

Taking sections 4.3.1 and 4.3.2 into account it should thus be possible to plan JointTrajectories for different planning-groups, save/load them and synchronize their execution.

4.4 Shortcomings of waypoint management

As mentioned in section 3.5 the developed methods enable the user to save, load and extend saved waypoint collections. What is not yet implemented is free editing of the saved waypoint collections, i.e:

- Removal of a waypoint at a arbitrary index of a waypoint collection
- Insertion of a waypoint at a arbitrary index of a waypoint collection

Implementing these two functionalities should however be fairly straight forward, since the waypoints are currently being saved as ROS PoseArray. Thus common array manipulations should suffice to implement the above mentioned functionalities.

4.5 Natural movements

When the search for a motion planning framework started, one concern was that methods such as cartesian path planning would lead to stiff, abrupt movements of the moneky robot. On a purely visual basis (which is all the living marmoset monkeys will have when interacting with the robot) the monkey robots movements look natural. It must however be noted that there are no path planning constraints in place, which enforce this. It is thus possible that MoveIt's compute_cartesian_method returns a JointTrajectory containing unnatural looking transitions between poses.

Blend_radius with Pilz Industrial Planner

If the monkey robot's execution of cartesian paths are perceived to be too stiff and abrupt, a possible solution could be to use the Pilz Industrial Motion Planner (Pilz IMP) instead of the Open Motion Planning Library (OMPL) currently in use (both of these live inside the MoveIt framework). The Pilz IMP offers the so called *sequence capability* which (similar to the compute_cartesian _path method in OMPL) enables the user to plan a path containing mulitple poses [17]. The sequence capability offers as input an argument called *blend_ radius*. This allows for two straight line segments of a path to be blended together via a round corner (as displayed in figure 4.1).

Making use of the sequence capability of the Pilz IMP could potentially make the monkey robots execution of multi pose trajectories look more natural.

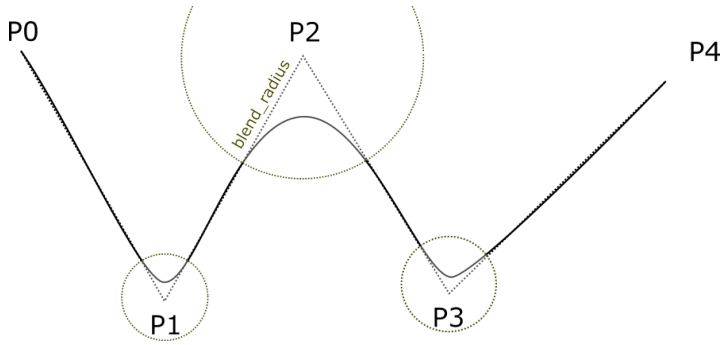


Figure 4.1: Softening of straight line path segments with the Pilz IMP (figure taken from [17])

Bibliography

[1] V. Marx, "Neurobiology: learning from marmosets," *Nature Methods*, vol. 13, pp. 911–916, 2016. [Online]. Available: https://doi.org/10.1038/nmeth.4036

[2] S. Biswas, "Why is the efficiency of low power motors lower than high power motors?" 06 2014.

[3] LibHunt, "Similar projects and alternatives to ros." [Online]. Available: https://www.libhunt.com/r/ros

[4] LibHunt, "Similar projects and alternatives to moveit2." [Online]. Available: https://www.libhunt.com/r/moveit2

[5] KST, "Kst x10 data sheet." [Online]. Available: https://cdn.shopify.com/s/files/1/0570/1766/3541/files/X10_SPECIFCATION.pdf?v=1675231593

[6] KST, "Kst a12-t data sheet." [Online]. Available: https://cdn.shopify.com/s/files/1/0570/1766/3541/files/A12-T.pdf?v=1673245327

[7] ROS, "Ros nodes." [Online]. Available: https://wiki.ros.org/Nodes

[8] L. Miller, "How to control a servo with raspberry pi." [Online]. Available: https://www.learnrobotics.org/blog/raspberry-pi-servo-motor/

[9] MoveIt!, "Moveit! move group node concepts." [Online]. Available: https://moveit.ros.org/documentation/concepts/

[10] I. S. (from ROS), "Moveit! move group class reference." [Online]. Available: https://docs.ros.org/en/jade/api/moveit_commander/html/classmoveit_commander_1_1move_group_1_1MoveGroupCommander.html

[11] LAAS, "Moveit dual arm planning concept." [Online]. Available: https://github.com/laas/moveit_dual_arm_planning/wiki/concept

[12] LAAS, "movegroupx (extended) overview." [Online]. Available: https://github.com/laas/moveit_dual_arm_planning/wiki/move_groupx

[13] LAAS, "Dual arm pick and place tutorial." [Online]. Available: https://homepages.laas.fr/jcortes/DualArmManipTuto/doc/pr2_tutorials/dual_arm_pick_place/doc/dual_arm_pick_place_tutorial.html

[14] Moveit, "Dual arm pose goals." [Online]. Available: https://github.com/ros-planning/moveit_tutorials/blob/indigo-devel/doc/pr2_tutorials/planning/src/move_group_interface_tutorial.cpp

[15] R. Answers, "Forum discussion about merging robottrajectories and handling collisions." [Online]. Available: https://answers.ros.org/question/370136/planning-for-a-dual-arm-robot-in-moveit/

[16] ROS, "Rospy message converter package summary." [Online]. Available: https://wiki.ros.org/rospy_message_converter

[17] ROS/Moveit, "Pilz industrial motion planner interface tutorial." [Online]. Available: https://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/pilz_industrial_motion_planner/pilz_industrial_motion_planner.html

[18] S. Sackett, "Solidworks to urdf using the sw2urdf plugin tutorial." [Online]. Available: https://www.youtube.com/watch?v=Id8zVHrQSlE&list=PL6TtDG40DrFpPf7OQagUNSW8dZMAvs5F0

[19] Moveit, "Moveit setup assistant tutorial." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html

[20] F. Reuleaux, "The kinematics of machinery (trans. and annotated by a. b. w. kennedy), reprinted by dover, new york (1963))," 1876.

Appendix A

RPP Setup

A.1 Operating system

Each ROS distribution is meant to be used with a specific Ubuntu distribution. For Ros Noetic the corresponding Ubuntu distribution is 20.04. For unknown reasons it proved impossible to install Ubuntu 20.04 on the RPP, thus the RPP-native "Raspberry Pi OS" (previously known as Raspbian) was installed.

With Raspbian installed a simple ROS environment was set up. This includes configuring the so called ROS-MASTER-URI on both devices connected to the ROS-network and setting the so called ROS-IP to the respective IP-Address of the device (see section B.2).

When work with the ROS-based framwork *MoveIt* started certain python packages could not be installed on Raspbian.

It was thus decided to switch to the Ubuntu 20.04 **Server** distribution for the RPP. This allowed for much easier installation of ROS packages and decreased CPU usage due to the headless (lack of GUI) setup.

The ubuntu server image used in this project is called "ubuntu-20.04.5-preinstalled-server-arm64+raspi.img.xz" and can be found on `https://old-releases.ubuntu.com/releases/20.04/`.

It proved to be very convenient to use the RPP imager software to flash the image, since it allows the user to specify a network access, which the RPP will already know and try to connect to when it first boots. Thus, one can directly ssh into the RPP over the specified network access.

A.2 Static IP Setup

Each time the RPP boots it can potentially be assigned a new IP-Address by the router.

Since the ROS-setup depends on the correct IP-Address being used, this frequent change of IP-Address has to be manually bypassed.

In order to mitigate this a static IP-Address can be set on the RPP. This can be done by adding a file with the following content in */etc/netplan* on the RPP:

```
# on the RPP in /etc/netplan
network:
    version: 2
    wifis:
        renderer: networkd
        wlan0:
            access-points:
                <WLAN-NAME>:
                    password: <WLAN-PWD>
            dhcp4: no
            optional: true
            addresses:
              - <DESIRED-IP>/24 #desired IP
            gateway4: <ROUTER-IP> # can be obtained with $ip r
            nameservers:
             addresses:
              - 8.8.8.8
```

Appendix B

ROS Setup

B.1 Overview

The development of ROS follows the yearly rhythm of Linux Ubuntu Distribution releases, with each ROS distribution being tailored to a specific Ubuntu Distribution. Starting from 2020 a new ROS version (ROS 2) was released which entailed certain paradigm changes in the areas of security, Real-Time Computing and Diverse Networks to name a few.

Since most knowledge and practical experience of not only the writer of this thesis but also the global ROS User community revolves around ROS1 and ROS1 offers all the functionality desired for this project it was decided to use ROS 1. More specifically, the latest and last distribution of ROS1; ROS Noetic.

A straight forward way to install Ros Noetic on the RPP is described on:

https://wiki.ros.org/noetic/Installation/Ubuntu.

For this thesis *ros-noetic-desktop-full* was installed on both the RPP and the desktop PC, but probably *ros-noetic-ros-base* would suffice for the headless RPP.

B.2 ROS Environment Variables Setup

To setup two devices (e.g a desktop PC and a RPP) the necessary shell commands look as follows (with the RPP as ROS-MASTER):

```
#for the RPP with IP-Address: <RPP_IP>
#execute these commands in the shell
export ROS_IP=<RPP_IP>
export ROS_MASTER_URI=http://<RPP_IP>:11311

#for the desktop PC with IP-Address: <PC_IP>
#execute these commands in the shell
export ROS_IP=<PC_IP>
export ROS_MASTER_URI=http://<RPP_IP>:11311
```

Appendix C

MoveIt Setup

C.1 Overview

Once ROS Noetic has been installed the next step is to install the matching MoveIt version. For this project, this was achieved by following the steps on:

https://ros-planning.github.io/moveit_tutorials/doc/getting_started/getting_started.html

C.2 MoveIt Setup Assistant

C.2.1 Launch Assistant

In order to use MoveIt with a custom robot a moveit_config package has to be generated by using the MSA. To initially launch the MSA, *cd* into your catkin workspace, run first:

```
source devel/setup.bash
```

and then run:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

This will startup the MSA. In the MSA, click on "Create New MoveIt Configuration Package" and select the URDF you have created with the URDF exporter.

C.2.2 URDF Exporter

The URDF needed for the moveit config package can be generated by the Solid Works URDF Exporter Plugin, provided there exists a STEP file of the robot model. This plugin enables the user to already configure certain aspects of the URDF file in SW, such as joint limits and which links belong to which arm. A windows executable installer for the plugin can be found on:

https://github.com/ros/solidworks_urdf_exporter/releases

Note that SW is only available for windows and macOS.

The whole process of generating the URDF file, from creating the joint axes in SW to importing it in the MSA is **well** described in a YouTube tutorial [18]. To avoid extending the length of this paper, kindly refer to the Youtube tutorial in order to see all necessary steps for generating the URDF file.

C.2.3 Configuring the moveit config package

A detailed description of all configurable features in the MSA can be found here:

https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html

The only critical parts needed to run the nodes developed in this project are the configuration of the the virtual joints, the planning groups and of the eef. These should be configured according to figures C.2,C.3 and C.1.

	End Effector Name	Group Name	Parent Link	Parent Group
1	monkey_left_hand	monkey_left_hand	L_Lower_Forearm_Link	monkey_left_arm
2	monkey_right_hand	monkey_right_hand	R_Lower_Forearm_Link	monkey_right_arm

Figure C.1: Configuration of the eef in MSA

	Virtual Joint Name	Child Link	Parent Frame	Type
1	fixed_frame	base_link	world	fixed

Figure C.2: Configuration of virtual joints in the MSA

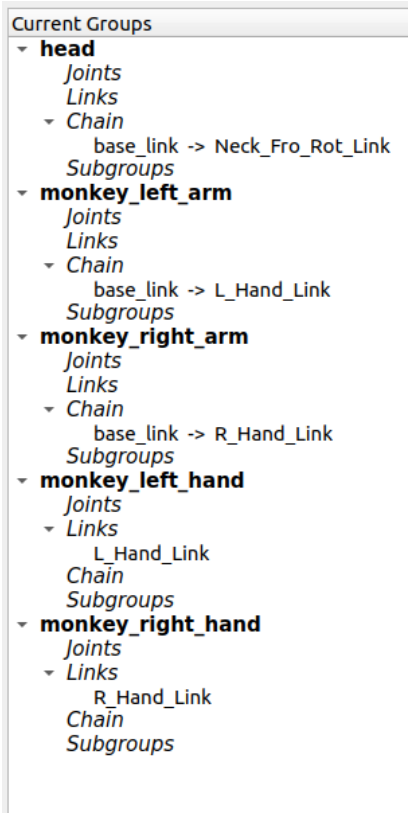


Figure C.3: Configuration of planning groups in the MSA

Once these configurations have been completed, the user can give the moveit config package a meaningfull name such as monkey_robot_moveit_config and save it in the /src folder of the catkin workspace.

Note that in the second step of the MoveIt Setup Assistant the Self-Collision Checking can be optimized by generating a collision matrix. "This searches for pairs of robot links that can safely be disabled from collision checking, decreasing motion planning time" [19].

C.2.4 Downloading developed packages

The packages developed during this project can be found on the following Github respository:

https://github.com/multiplexcuriosus/monkey_robot_codebase

Make sure that the monkey_interface and monkey_complete package are present in the src folder of the catkin workspace of your PC and the package monkey_listener is present in the src folder of the catkin workspace of the RPP.

At this point you should have generated your own moveit-config pkg and it should also be present in the src folder of the catkin workspace of your PC. Whether the setup works with the monkey_robot_moveit_config package developed during this project is discussed in the ReadMe of the Github Repository from which the monkey_interface package was downloaded.

C.2.5 Building all packages

In the workspaces on your PC and on the RPP run:

```
catkin build or
```

or

```
catkin build <name-of-single-pkg>
```

if you want to only build one package (which might be desirable since building all ca. 50 MoveIt related packages takes 10 *min*).

C.2.6 Launching the nodes

In order to launch mulitple ROS nodes at once, possibly with mulitple input arguments for each, ROS makes use of so called launch files. These files (usually in XML format) contain all the information ROS needs to know about which nodes to launch, in which order and with which input arguments.

For our convenience the MSA has generated a demo launch file called *demo.launch* inside the generated moveit config package. This demo.launch file among other things contains instructions to launch a Rviz node and a MoveIt motion planning node.

In order to make use of the nodes written in this project we first run (in a terminal T_1):

```
roslaunch <moveit-config-pkg> demo.launch
```

This will launch multiple nodes, including Rviz and the MoveIt motion planning plugin. You should see an Rviz window appear with a panel labeled MotionPlanning. Then we open a second terminal T_2, navigate into the catkin workspace, source the setup.bash file and run:

```
rosrun monkey_interface monkey_interface.py
```

In T_2 there should soon appear some options and a prompt to choose one of the options. These options refer to the planning methods described in section E.1.

Appendix D

Controlling the Servos

D.1 Overview

A python script *joint_ control.py* was written to control the motors. This script can be launched either as python3 script in the shell or as node inside the ROS network. The code examples in the remainder of this chapter use the shell method.

To run the script as a ROS node simply replace:

```
python3 <script> <arguments>
```

with:

```
rosrun monkey_listener <script> <arguments>
```

where monkey_listener is the name of the package the <script> is contained in. Running the script as ROS node will only work if the ROS infrastructure is setup as described in section B.2.

D.1.1 Control formats

The joint-control-script mentioned above offers different ways (control formats) to control the servo motors.

Duty Cycle

The first control format is setting the duty cycle of a servo motor. The KST-A12 can be moved to the positions -50°/0/50° by providing the corresponding pulse lengths 1000µs, 1500 µs, 2000 µs. Using 50Hz this corresponds to duty cycles of 4.5%, 7.5% and 10.5%.

Range Presets

The next control format originates from the following two problems:

1. Finding a mapping between duty cycle ($\in [4.5, 10.5]$) and minimal/maximal rotation of a joint. Minimal and maximal here don't merely correspond to the range permitted by the metal body joints of the robot, but instead to the range resulting from all the factors listed in section 2.1.8.
2. The inconsistent thread attachment mentioned in section 2.1.6.

To address these issues a second system was introduced, where each joint object in the code stores its duty cycles values for certain joint states.

Presets		
ID	Param	Description
DT_{min}	0	Joint Range Minimum
DT_{middle}	0.5	Joint Range Middle
DT_{max}	1	Joint Range Maximum
DT_{def}	def	Resting position

Table D.1: Presets introduced to facilitate testing.

Now we can for each joint J experimentally determine the duty cycles DT_{min}, DT_{max} which bring J to his minimal and maximal rotation. In the python scripts joint_control.py and monkey_listener.py we can then save DT_{min} and DT_{max}. Additionally we can save the duty cycle value corresponding to the middle DT_{min} and DT_{max} and a DT_{def} value, which stores the duty cycle of the position the joint has when its corresponding planning group is in its default position. The vales in the column *Param* in table D.1 are the possible input arguments for the methods described in sections D.3,...

PiGPIOFactory()

As mentioned in section 2.4.2 there was a lot of jitter initially, which could be eliminated by using the PiGPIOFactory() factory to control the servos. The servo control functions offered by the gpiozero library accept as input a *float* between -1 and 1. Since this method reduced the jitter significantly it is used now to control the servos. In order to build on top of the Range Presets, mapping functions were created to map from duty cycle to [-1,1].

D.2 Prerequisites for launch of joint control node

Running the node as python3 script from the shell

The "pigpio daemon" has to be running. He can be started by executing the following command in a shell:

```
sudo pigpiod
```

Running the node as ROS node with rosrun

The prerequisites are:

- On the ROS master device (typically the RPP) the command **roscore** was executed in a terminal. This starts the ROS network.
- The ROS environment variables were setup according to section B.2
- The pigpio daemon is running

D.3 Set Duty Cycle

Set the duty cycle on a joint by running:

```
python3 monkey_control_node.py raw NH 4.5
```

where *raw* specifies that the control format is duty cycles, *NH* is the target joint and 4.5 is the duty cycle to be written to the NH servo motor.

The servos can be set to the min, half, max or def position by using the command:

```
python3 monkey_control_node.py single NH 0
```

where *single* refers to the mode of setting a min,half,max state and *0* refers to the min state ($\{min,\ half,\ max\} := \{0,\ 0.5,\ 1\}$).

In order to set all joints to their def position run:

```
python3 monkey_control_node.py alldef
```

D.4 Monkey_listener

To enable control of the physical robot the monkey_listener package launches a node, which does two things.

- It it subscribed to the joint_states topic.
- When the subscriber reads new joint-state data, it uses the exact same methods as the joint_control.py script to write the received joint-states to the servo motors.

Thus this node must be running whenever we wish to control the physical robot.

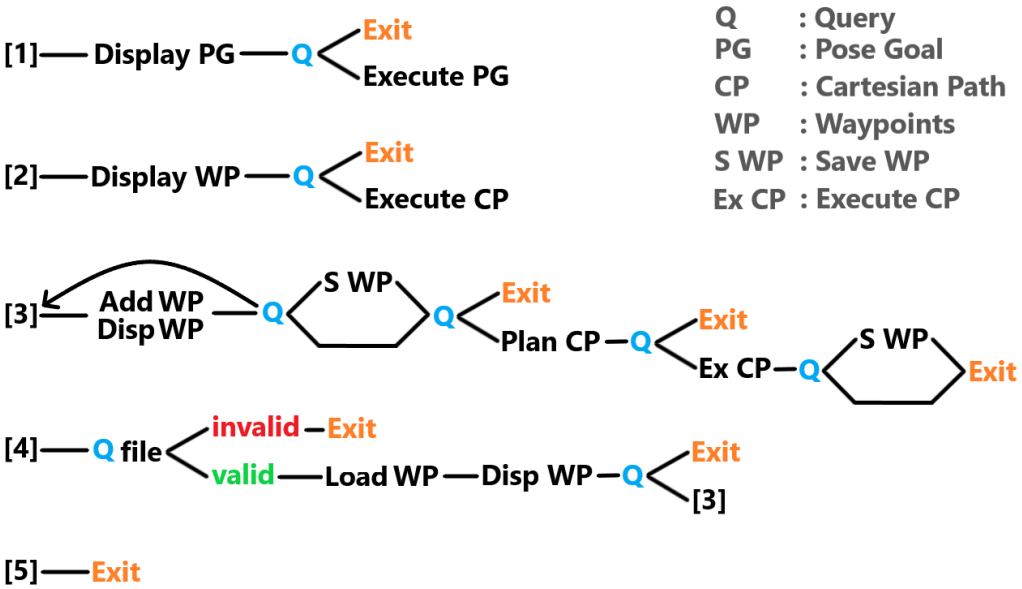
Appendix E

Monkey Interface

E.1 Overview

During this project a ROS node called monkey-interface was created which allows the user to use the functionalities described in section 3.5.

In order to make use of the functionalities, all relevant nodes must have been launched according to the steps described in section C.2.6. The following figure depicts the control flow of the monkey interface.



E.2 Usage notes regarding mode [3]

Out of the box the eef of the monkey arms cannot be moved with the IM, since there are not enough DOF per arm to allow free movement/manipulation of the eef. In order to move the eef with the IM, the "Approx IK Solutions" options box has to be ticked, such that approx. IK solutions are allowed. If the eef becomes hard to move during workflow, it might be the case that this option was unticked (e.g after a Rviz reset) *without* the tick disappearing. Simply untick and tick the option again in such cases. This does not affect the position tolerance of computing cartesian paths.

Appendix F

Abbreviations

F.1 Hardware

The following table lists the abbreviations used for all joints.

ID	Joint Description
W	Wrist
SH	Shoulder horizontal
EB	Elbow
SL	Shoulder Lateral
H	Hand
SF	Shoulder frontal

Note that in the text, the joint ID's will often have an 'L' or 'R' prepended to indicate the left or right arm of the robot monkey.

Appendix G

Acknowledgments

I would like to express my gratitude towards:

- Joris Gentinetta, for his patient mentoring and skillful collaboration
- The D-ITET Wersktatt for the fabrication and adaptation of the robot
- Simon Steffens for the fabrication of the wheels
- Wolfger von der Behrens for ordering the parts

Glossary

end effector Part of a robotic arm, usually at the end of a kinematic chain. Often contains a gripper to grasp objects. 20

kinematic chain Assembly of rigid bodies connected by joints to provide constrained motion [20]. 10

link Word used in the context of ROS and URDFs, referring to a geometrically isolated physical part of the robot body. The head of the monkey robot for example consists of two links. 16

pose Message type from ROS, contains a position (3D vector) and an orientation (4d Quaternion). 19

screw locking agent Liquid "glue" which can be applied to screws up to M36 in order to lock the internal threads position in the external thread. For this project "TOOLCRAFT Schraubensicherung mittelfest" was used. 11

Spiderwire Fisherman's line with a tear resistance of 38.1 kg/0.33 mm, bought at *fischen.ch*. 8