Imperial College London

Mechatronics in Medicine Laboratory

A modular ROS platform for virtual reality assisted teleoperation of KUKA iiwa robot manipulators and ReFlex TakkTile robotic hand supporting the Leap Motion gesture tracking device

Project Report

Author:	Alexander Koenig
Supervisors:	Dr. Fabio Tatti
	Dr. Riccardo Secoli
Advisor:	Prof. Dr. Ferdinando Rodriguez y Baena
Dates:	01.04.2019 - 12.09.2019

Glossary

GUI Graphical User Interface. 19, 20, 33, 46–48

HMD Head Mounted Display. 9, 51

KST KUKA Sunrise Toolbox. 17, 19, 21, 34, 59, 60

PTP Point to point. 17–19, 21, 31, 32, 34, 59

RGB-D Red-Green-Blue-Depth. 4, 9, 51, 52, 54, 55

ROS Robot Operating System. 1, 3, 9, 10, 17–22, 27, 31–35, 37, 38, 41, 43, 45, 50

VR Virtual reality. 10, 51

List of Figures

1.1 Overview of robotic rig . . . 9
1.2 Hardware architecture . . . 11
1.3 Software Architecture: Leap Motion control . . . 12
1.4 Networking setup KUKA . . . 14
1.5 Networking setup robotic hand . . . 14

2.1 KUKA Sunrise Toolbox communication scheme [7] . . . 17
2.2 PTP Motion [1, p. 304] . . . 18
2.3 LIN Motion [1, p. 304] . . . 18
2.4 Feedback of ROS services . . . 20
2.5 Graphical user interface . . . 20
2.6 Workspace parameters (side view) [2, p. 36] . . . 24
2.7 Workspace parameters (top view) [2, p. 36] . . . 24
2.8 Workspace plot (center line [2,-1,0], opening angle 50 deg) . . . 25
2.9 Workspace plot (center line [1,0,0], opening angle 120 deg) . . . 25
2.10 Coordinate systems [1] . . . 26
2.11 Joint convention [1] . . . 26
2.12 Calculation of workspace center . . . 28
2.13 Calculation of home position . . . 28
2.14 Home position with center line [1,0,0] . . . 29
2.15 Home position with center line [-1,0,0] . . . 29
2.16 Plot of workspace custom home position [-45, 20, 0, -100, 0, 40, 0] . . . 30
2.17 Hand coordinate system [5] . . . 36
2.18 Coordinate systems of left hand right hand [5] . . . 36
2.19 Leap Motion coordinate system [5] . . . 37
2.20 Coordinate transformations overview . . . 37
2.21 Position of Leap Motion in KUKA base frame . . . 38
2.22 Transformation to virtual Leap Motion . . . 39
2.23 Transformation to desired KUKA tool pose . . . 40
2.24 Frames published by leap_kuka node viewed in RViz . . . 41

3.1 Human flexion angles [6] . . . 44

4.1 Leap Motion diagnostic visualizer . . . 49

4.2 Leap Motion workspace [8] 49

5.1 Workspace as viewed inside the Oculus 51
5.2 Camera pose calculation in KUKA base frame 52
5.3 Tf tree when camera_apriltag and a KUKA control node running 53
5.4 RViz showing fused RGB-D data 54

6.1 Network architecture of GGCNN [4] 55
6.2 RGB view 56
6.3 Grasping point prediction 56
6.4 RGB view 56
6.5 Grasping point prediction 56
6.6 Enhanced GGCNN Overview 57

List of Tables

1.1 Networking settings 15
1.2 Folder Structure of GitHub repository 15

2.1 Matlab scripts of the KUKA control node 17
2.2 ROS services provided by the KUKA control nodes 19
2.3 ROS messages published by the KUKA control node 21
2.4 KUKA configuration parameters 22
2.5 Leap Motion configuration parameters 35
2.6 Transformations published by leap_kuka node 38

Contents

List of Figures **3**

List of Tables **5**

1 Introduction **9**
1.1 Hardware Components . . . 9
1.2 Software Components . . . 10
1.3 Installation . . . 13
1.4 Networking Setup . . . 14
1.5 Folder Structure . . . 15

2 KUKA Control Node **17**
2.1 General . . . 17
2.1.1 Scripts Overview . . . 17
2.1.2 KUKA Motion Types . . . 18
2.1.3 Services of Control Scripts . . . 19
2.1.4 Topics of Control Scripts . . . 21
2.1.5 KUKA Configuration File . . . 22
2.1.6 Workspace Definition . . . 23
2.1.7 Enforcing Workspace Boundary . . . 26
2.1.8 Home Position Calculation . . . 27
2.1.9 Custom Home Pose . . . 30
2.1.10 Control Loops . . . 31
2.1.11 Running Control Scripts . . . 33
2.2 Leap Motion . . . 35
2.2.1 Leap Configuration File . . . 35
2.2.2 Coordinate Systems . . . 36
2.2.3 Coordinate Transformations . . . 37

3 Robotic Hand Node **43**
3.1 General . . . 43
3.2 Leap Motion . . . 43

4 Tips and Tricks **45**
4.1 Leap Rig Demo . . . 45

4.2 Troubleshooting 48
4.3 Leap Motion Details 49
4.4 ROS Commands 50

5 Vision **51**
5.1 Virtual Reality 51
5.2 Visual Fiducial 52
5.3 RGB-D Fusion 54

6 Autonomous Grasping **55**

7 Future Work **59**

Bibliography **61**

1 Introduction

In this report a modular ROS platform for the teleoperation of a KUKA iiwa (7 R800 or 14 R820) robot manipulator and an attached ReFlex TakkTile robotic hand is presented. The system supports the control of both devices using the Leap Motion gesture tracking device. It also features the integration of a depth camera and a virtual reality HMD. The following sections give a brief overview of the involved hardware and software components. Since this is a modular system not all pieces of hardware (or software) are needed to run a specific functionality of the system.

1.1 Hardware Components

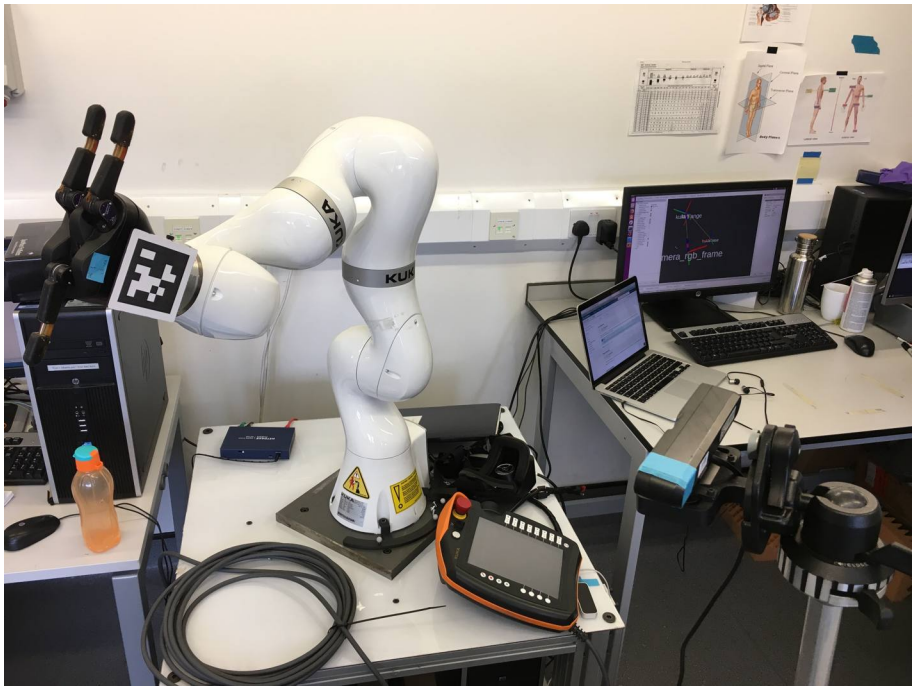

Figure 1.1: Overview of robotic rig

Figure 1.2 shows an overview of the hardware architecture of the robotic rig. Ethernet cables should be category 5 or higher. The RGB-D camera is mounted on a tripod.

- KUKA iiwa 7 R800 or 14 R820
- ReFlex TakkTile Robotic Hand
- Leap Motion
- Oculus Rift DK2
- Asus Xtion Pro Live
- Switch
- 3 ethernet cables

1.2 Software Components

The system was developed and tested using the following software.

- Ubuntu Xenial 16.04 LTS
- Python 2.7
- Numpy 1.11.0
- MATLAB Version 9.6 (R2019a)
- MATLAB Instrument Control Toolbox Version 4.0 (R2019a)
- MATLAB Robotics System Toolbox Version 2.2 (R2019a)
- MATLAB Robotics System Toolbox Interface for ROS Custom Messages
- ROS Kinetic (Desktop Install recommended)
- KUKA MatlabToolboxServer on KUKA Robot Controller. See user guide of KUKA Sunrise Toolbox
- Only for grasp prediction: CUDA V10.1.168 (currently not needed)
- Only for grasp prediction: Conda 4.6.14 (installed miniconda2)
- Only for grasp prediction: Conda environment with dependencies in grasping/env/environment.yaml

Figure 1.3 shows an overview of the ROS network used for the control of the robotic rig using the Leap Motion tracking device. The ROS nodes for the integration of the VR display and the depth camera into the platform are intentionally not displayed in Figure 1.3 since these nodes run completely independently from the control of the robotic rig using the Leap Motion.

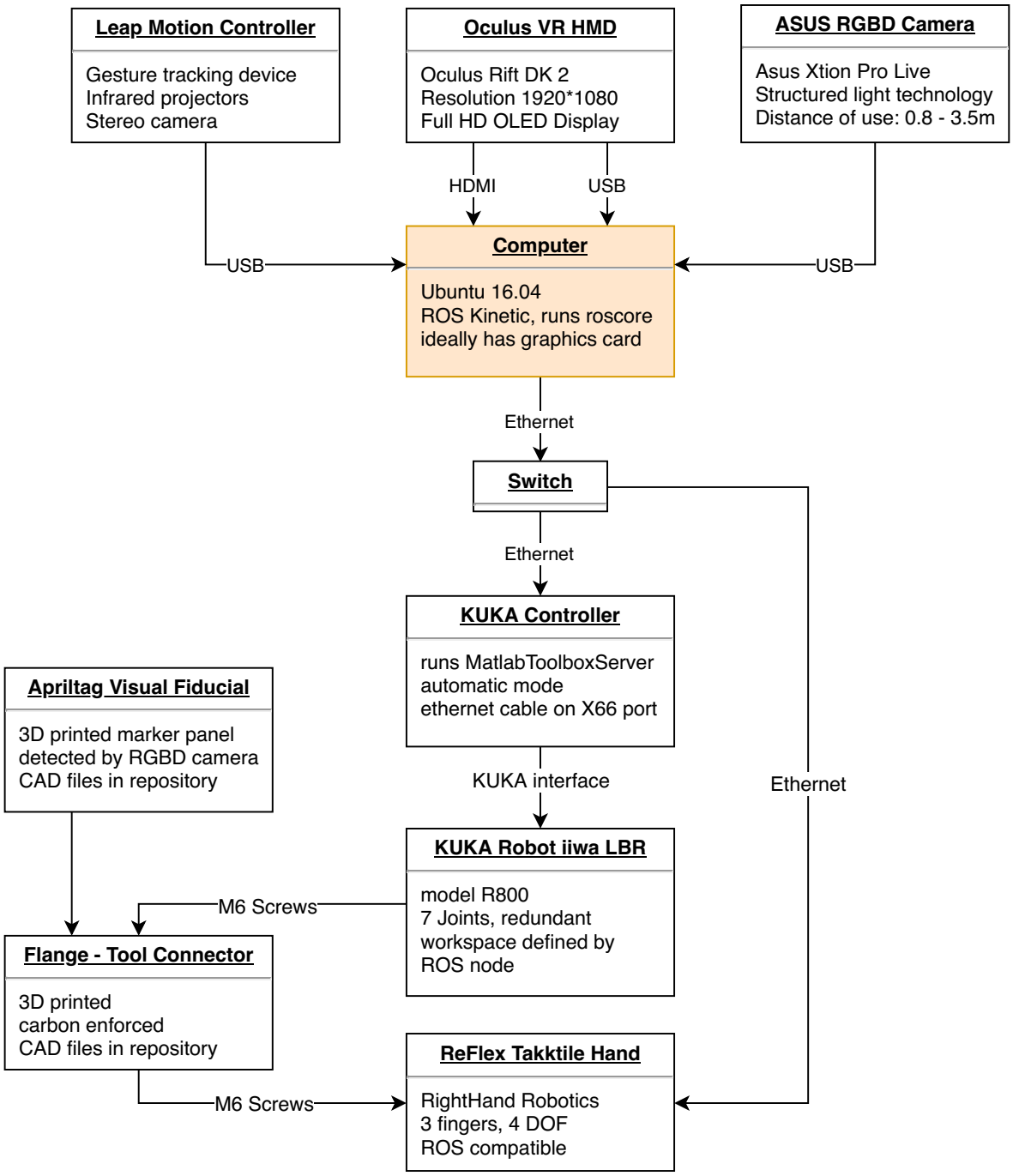

Figure 1.2: Hardware architecture

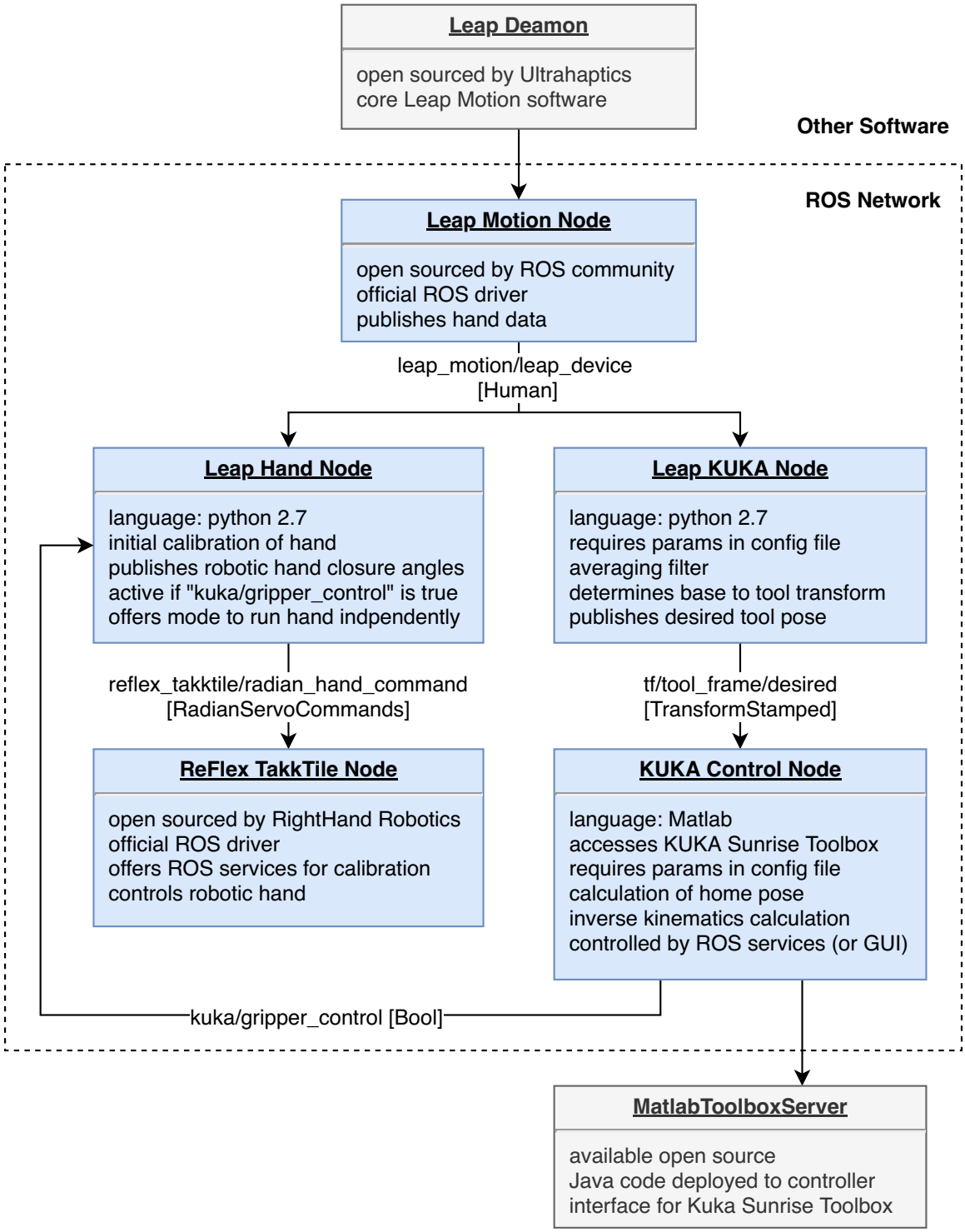

Figure 1.3: Software Architecture: Leap Motion control

1.3 Installation

This is a step by step guide on how to install the software taken from the README file of the GitHub repository. Note that the above software components first need to be installed.

1. Install all of the above software components.
2. Initialize a clean catkin workspace. Open a terminal and type

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

3. Clone this repository with all its submodules in the src folder of your workspace

```
git clone --recursive
git@github.ic.ac.uk:KukaProject/alex_code.git
```

4. Install all dependencies. You might need to setup rosdep first

```
rosdep update
rosdep install --from-paths src --ignore-src -r -y
```

5. Build workspace with catkin build

```
cd ~/catkin_ws
catkin build
```

6. Remember to source the setup.bash file or add it to your .bashrc

```
source /opt/ros/kinetic/setup.bash
or
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

7. Call `rosgenmsg("~/catkin_ws/src/kuka/kuka_msgs")` in Matlab command window and follow onscreen instructions. See further instructions here

1.4 Networking Setup

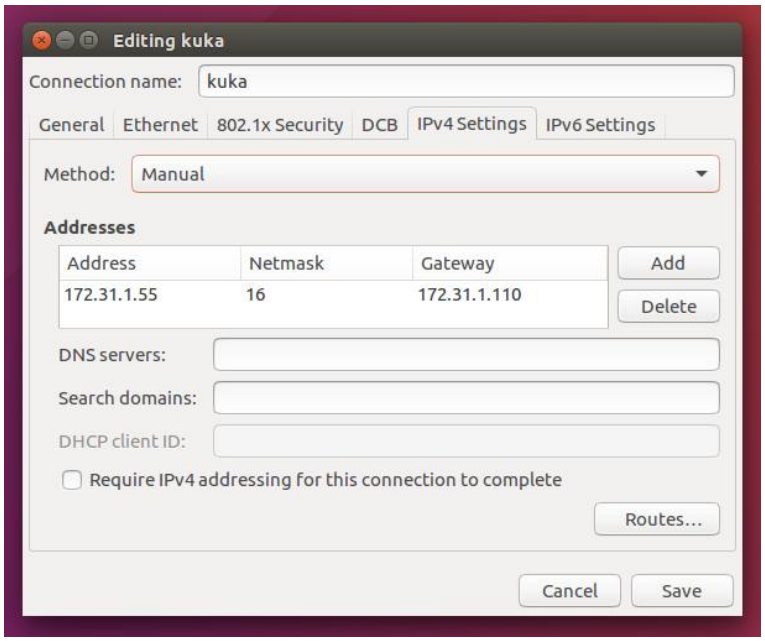

Figure 1.4: Networking setup KUKA

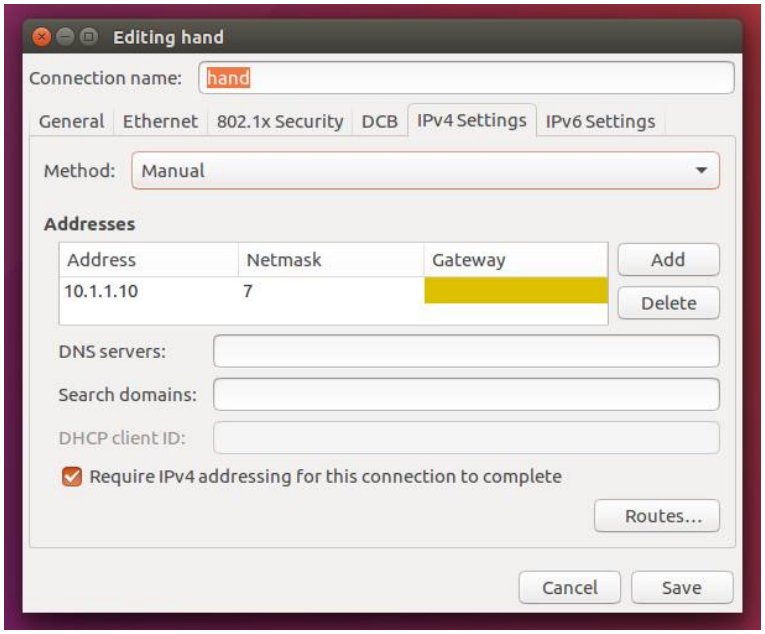

Figure 1.5: Networking setup robotic hand

Setting	KUKA	Robotic Hand
Address	172.31.1.55	10.1.1.10
Netmask	16	254.0.0.0
Gateway	172.31.1.110	0.0.0.0

Table 1.1: Networking settings

Remember to check "Require IPv4 addressing" button for the robotic hand. Also make sure your ethernet connection shows up as “eth0”. If it does not this link might help. Please view the Robotic Hand’s documentation for more details. In their instructions you can skip all sections on cloning and building the drivers as they will already be included in this package. Before you use the networking connections make sure you actually connect to them by clicking on the connections in the networking center.

1.5 Folder Structure

Folder Name	Contents
cad	CAD files of 3D printed parts
common	library of common functions used by leap_hand and leap_kuka
docs	handbook
grasping	code for autonomous grasping prediction (work in progress)
kuka	code for controlling the KUKA robot
leap_hand	code to interface between Leap Motion node and robotic hand node
leap_kuka	code to interface between Leap Motion node and KUKA node
leap_rig	files to start Leap Motion control of KUKA and robotic hand
modules	third party software (created upon recursive clone)
vision	all vision related code (camera and virtual reality)

Table 1.2: Folder Structure of GitHub repository

2 KUKA Control Node

The KUKA control node essentially provides a ROS wrapper of the Kuka Sunrise Toolbox presented in [7]. The Kuka Sunrise Toolbox is a Matlab Toolbox which interfaces with a Java application called MatlabToolboxServer on the KUKA robot controller. The toolbox provides intuitive Matlab functions to control the robot. Figure 2.1 shows an overview of the KST architecture.

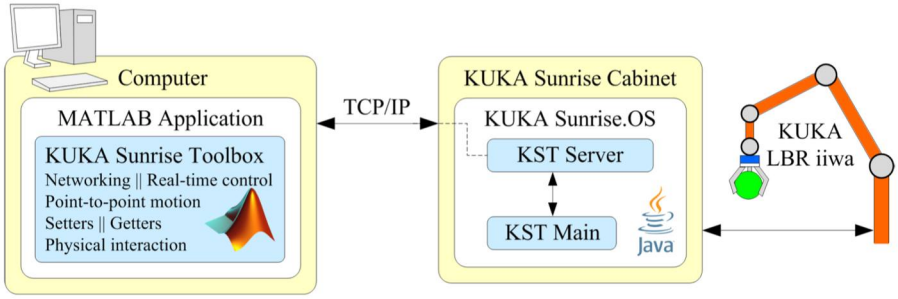

Figure 2.1: KUKA Sunrise Toolbox communication scheme [7]

2.1 General

2.1.1 Scripts Overview

The KUKA control package offers four main scripts in the directory `kuka/kuka_control/scripts`. The folder `functions` contains all the relevant functions that are used within these four scripts.

Script Name	**Functionality**
ptpController.m	ROS interface for PTP control of the robot
realTimeController.m	ROS interface for real-time control of the robot
simpleController.m	short script to send the robot to a joint configuration
workspacePlotter.m	script to plot a workspace defintion

Table 2.1: Matlab scripts of the KUKA control node

2.1.2 KUKA Motion Types

The two motion types used in this software are **PTP movements** and **Direct Servo** motions. The `ptpController.m` script uses only PTP movements, whereas the `realTimeController.m` script uses both methods (movement to the home position, zero position, the mirroring pickup point in PTP mode, otherwise in real-time mode). Direct Servo is a functionality issued by KUKA to allow for "soft" real-time control of the robot in joint space. The documentation about the Direct Servo motion type is quite sparse, unfortunately. Any motions executed with the Direct Servo functionality are referred to as real-time motions in this report.

This is a description of PTP movements taken from the KUKA manual. "The robot guides the TCP along the fastest path to the end point. The fastest path is generally not the shortest path in space and is thus not a straight line. As the motions of the robot axes are simultaneous and rotational, curved paths can be executed faster than straight paths. PTP is a fast positioning motion. The exact path of the motion is not predictable, but is always the same, as long as the general conditions are not changed" [1, p. 303].

Below you can get a graphical intuition of the PTP motion type as opposed to the linear motion type (which is not used in the software). In the LIN motion type the "robot guides the TCP at the defined velocity along a straight path in space to the end point" [1, p. 304].

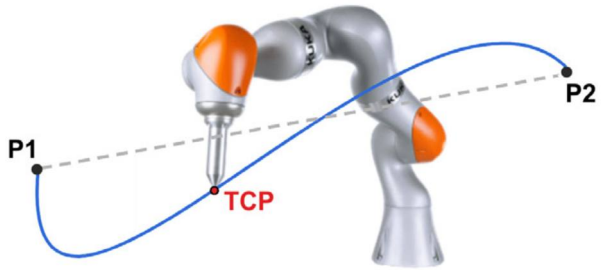

Figure 2.2: PTP Motion [1, p. 304]

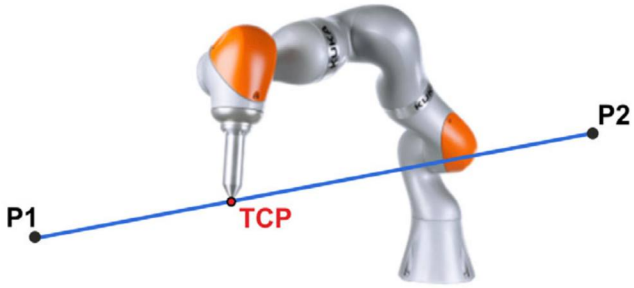

Figure 2.3: LIN Motion [1, p. 304]

The PTP and real-time ROS nodes use the functions `movePTPJointSpace(jPos, relVel)` and

`sendJointsPositionsf(jPos)` that are offered by the KST to move the robot. The `jPos` variable is a 1x7 cell array of target joint positions. The `relVel` variable specifies the relative motion execution speed (i.e. a value from 0 to 1, with 1 indicating maximum speed). Please refer to the KST's user manual in their GitHub repository for more information.

2.1.3 Services of Control Scripts

Both the `ptpController.m` and the `realTimeController.m` script have exactly the same ROS interface. They can both be controlled with ROS services or a Graphical User Interface. Table 2.2 shows a list of the services provided by the script. When the ROS services are called (like shown in Listing 2.1) the user will receive a feedback message, if the service call was successfully executed (also see Figure 2.4).

Service Name	**Service Type**	**Description**
kuka/exit_app_emergency	kuka_msgs/exitAppEmergency	terminate Matlab
kuka/exit_app	kuka_msgs/exitApp	shutdown control script
kuka/kuka_control	kuka_msgs/setKukaControl	switch KUKA control on/off
kuka/gripper_control	kuka_msgs/setGripperControl	switch gripper control on/off
kuka/scaling_factor	kuka_msgs/setScalingFactor	scale translation

Table 2.2: ROS services provided by the KUKA control nodes

```
rosservice call kuka/exit_app_emergency
rosservice call kuka/exit_app
rosservice call kuka/kuka_control true
rosservice call kuka/gripper_control true
rosservice call kuka/scaling_factor 0.7
```

Listing 2.1: Usage of ROS services

All services become available once either the `ptpController.m` or the `realTimeController.m` Matlab script is run. Use the `exit_app` service to shut down the connection to the robot controller and terminate Matlab. The `exit_app_emergency` service will terminate Matlab immediately without shutting down the robot connection. The `kuka_control` service is used to enter either PTP or real-time control mode (depending on which script is running). With the `gripper_control` service the boolean which is published on the `kuka/gripper_control` ROS topic can be changed. The fact that users have to activate the control of the KUKA and the gripper once the control script is running can be seen as a safety measure to avoid unintentional motions. The range in which movements can be scaled is 0.1 to 1.5. It is not encouraged to change this to ensure safety.

The Graphical User Interface in Figure 2.5 can also be used to control the software. Each service corresponds to a button on the GUI (except the "Reset Scaling Factor" button). The slider in the

```
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC: ~
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC:~$ rosservice call /kuka/kuka_control true
success: True
message: "Activated KUKA control. Listening to /tf topic."
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC:~$ rosservice call /kuka/scaling_factor 0.8
success: False
message: "Real-time control must be switched off to set scaling factor."
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC:~$ rosservice call /kuka/kuka_control false
success: True
message: "Deactivated KUKA control. Not listening to /tf topic."
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC:~$ rosservice call /kuka/scaling_factor 1.7
success: False
message: "Scaling factor must not be larger than 1.5."
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC:~$ rosservice call /kuka/scaling_factor 0.8
success: True
message: "Setting scaling factor to 0.8."
kukamaster@kukamaster-HP-Compaq-8100-Elite-CMT-PC:~$ rosservice call /kuka/exit_app
success: True
message: "Shutting down robot connection. Exiting the application."
```

Figure 2.4: Feedback of ROS services

GUI makes the scaling factor continuously adjustable. The "ROS Control" buttons in the KUKA and Gripper panels will turn green once the respective functionality is actually available (also turns green if activated through service call). This gives an intuitive graphical feedback in which state the software is currently in.

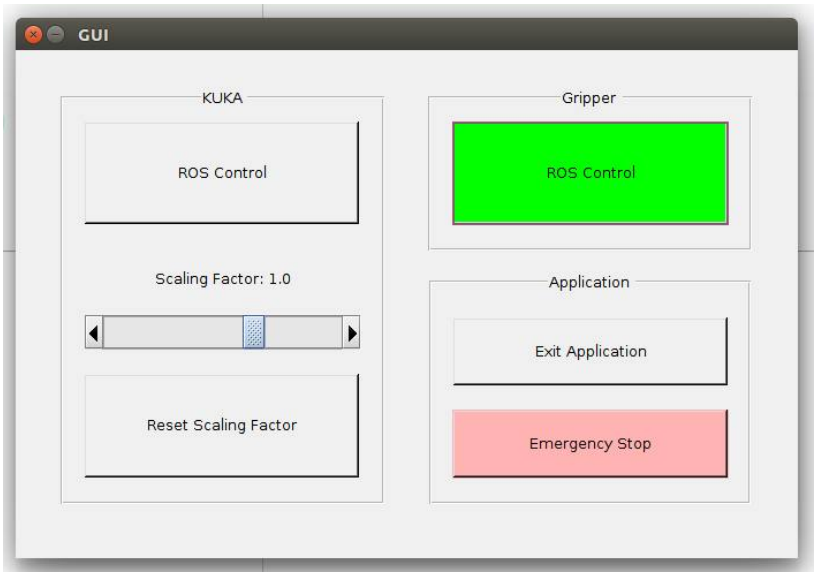

Figure 2.5: Graphical user interface

2.1.4 Topics of Control Scripts

When either the `ptpController.m` or the `realTimeController.m` is run the below ROS topics become available and data about the robot state is published. Especially notable is the `kuka/gripper_control` topic, which indicates the status of an attached gripper (see chapter 3). All transforms published are in the KUKA base frame and can be conveniently accessed in the ROS tf tree.

Note that all messages of the `kuka_msgs/` type and the `tf/kuka/flange_frame/measured` can not be provided while the robot moves in real-time mode. The topics will still exist, but while real-time movements are executed no new messages will be published to these topics. The reason for this is that when in real-time mode, the KST does not allow to request the robot state. This only works reliably when in PTP mode. Possibly, the KST will be updated in the future with functions to allow for this (see chapter 7).

Topic Name	**Message Type**
kuka/gripper_control	std_msgs/Bool
kuka/moving_rt	std_msgs/Bool
kuka/moving_ptp	std_msgs/Bool
tf/kuka/flange_frame/home	geometry_msgs/TransformStamped
tf/kuka/tool_frame/home	geometry_msgs/TransformStamped
tf/kuka/flange_frame/measured	geometry_msgs/TransformStamped
kuka/flange_force	geometry_msgs/Vector3Stamped
kuka/flange_torque	geometry_msgs/Vector3Stamped
kuka/joint_positions	kuka_msgs/jointPositions
kuka/joint_torques_external	kuka_msgs/jointTorquesExternal
kuka/joint_torques_measured	kuka_msgs/jointTorquesMeasured

Table 2.3: ROS messages published by the KUKA control node

To move the robot using the KUKA control node a stream of ROS `geometry_msgs/TransformStamped` messages from `kuka/base_frame` to `kuka/tool_frame/desired` should be published to the ROS network. This is the only topic the ROS node subscribes to.

2.1.5 KUKA Configuration File

The following parameters have to be set in the configuration file called `kuka_params_ptp.yaml` or `kuka_params_rt.yaml` depending on the control script you are using. The parameters must be uploaded to the ROS parameter server before running the control scripts. This can be done with the included launch files `kuka_params_ptp.launch` or `kuka_params_rt.launch`. The parameters's units are meters, degrees and seconds.

Mode	Parameter Name	Default	Note
both	robot	LBR7R800	LBR7R800 or LBR14R820
both	flange	MF_elektrisch	MF_elektrisch, MF_pneumatisch, MF_IO_pneumatisch or MF_touch_pneumatisch
both	ip	172.31.1.147	robot ip
both	center_line	[1, -1, 0]	z must be zero
both	opening_angle	50.0	range (0, 180)
both	z_lower_limit	0.2	range (0.1, z_upper_limit)
both	z_upper_limit	0.6	range (z_lower_limit, 1.14)
both	inner_sphere_limit	0.5	range (0.4, outer_sphere_limit)
both	outer_sphere_limit	0.7	range (inner_sphere_limit, 0.8)
both	tool_length	0.106	range [0, inf)
both	time_out	0.2	move home after time_out
both	home_pos	[-45, 30, 0, -80, 0, 70, 0]	disregard if use_home_pos false
both	use_home_pos	false	use home_pos param or not
rt	velocity_ptp_slow	0.15	range (0, 1)
rt	velocity_ptp_fast	0.25	range (0, 1)
ptp	velocity_ptp	0.15	range (0, 1)
ptp	joints_thresh	0.5	if below not moving

Table 2.4: KUKA configuration parameters

The `time_out` parameter specifies after what time the robot moves back to its home position if the KUKA node does not receive any more new tf messages. The node always measures the time from the last received ROS time stamp. If a "new" message is received (i.e. time stamp differs from previous message) the timer is reset. If you set the `time_out` parameter to a high number (e.g. 999999 seconds) the robot will stay in position that corresponds to last message and not move home until the set `time_out` is reached. When you publish new tool transforms to the KUKA control node, be sure that the pose is close to the one when you left off. In future work the robot should rather do a PTP motion to the new pickup point (see chapter 7).

In the current state of the software it is only possible to define the tool length and not a full transformation matrix from flange to tool. This is to keep calculations of the home pose simple.

The tool length is measured along the flange's Z axis. Since the tool is fixed to the flange, the rotation of flange and tool is identical. If a tool length of 0 is set the flange frame will be controlled.

It is the operators responsibility to ensure that using the provided parameters the robot can reach all positions in the workspace. If the robot can not reach a position it will stay in its previous pose.

2.1.6 Workspace Definition

Since the working envelope of the KUKA robot follows the shape of a hollow sphere (grey area in Figure 2.6) it was decided that the workspace boundary should be a segment of this spherical shape. The workspace of the robot is defined by the parameters `center_line`, `opening_angle`, `z_lower_limit`, `z_upper_limit`, `inner_sphere_limit` and `outer_sphere_limit`. Figures 2.6 and 2.7 show an overview of what the individual workspace parameters mean.

It is encouraged to visualise the workspace with the Matlab script `workspacePlotter.m` before editing the configuration file. Figure 2.8 shows a workspace plot. For the plot many random positions are sampled and then constrained using the function `getConstrainedPosition.m`, which is also used in the control scripts. The visualisation also shows the calculated workspace center and the home pose of both the tool and flange frame. The tool tip (center of frame) always coincides with the workspace center (red dot in Figure 2.8) when the robot reaches its home position. Also view Figure 2.9 to see the visualisation of another possible workspace.

Please see the below conventions for coordinate systems (Figure 2.10) and joint conventions (Figure 2.11) which are used throughout this software. The KUKA manual describes the joint convention as follows. "The positive direction of rotation of the robot axes can be determined using the right-hand rule. Imagine the cable bundle which runs inside the robot from the base to the flange. Mentally close the fingers of your right hand around the cable bundle at the axis in question. Keep your thumb extended while doing so. Your thumb is now positioned on the cable bundle so that it points in the same direction as the cable bundle runs inside the axis on its way to the flange. The other fingers of your right hand point in the positive direction of rotation of the robot axis" [1, p. 87].

For labelling axes the standard "XYZ-RGB" convention is used in this report.

1. X axis corresponds to red color
2. Y axis corresponds to green color
3. Z axis corresponds to blue color

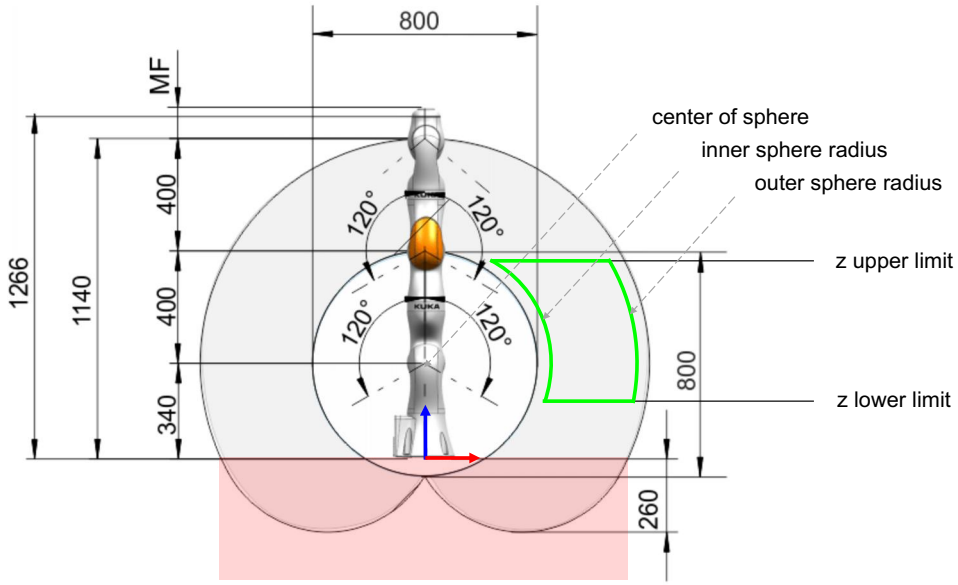

Figure 2.6: Workspace parameters (side view) [2, p. 36]

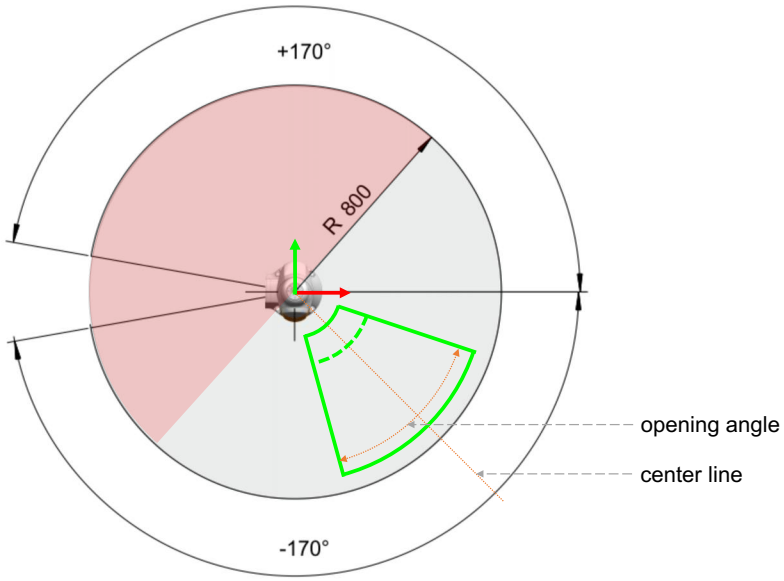

Figure 2.7: Workspace parameters (top view) [2, p. 36]

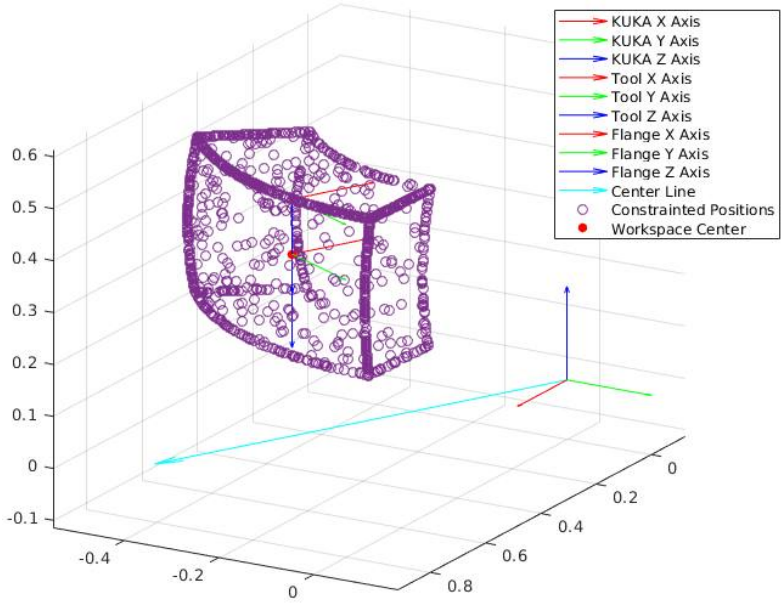

Figure 2.8: Workspace plot (center line [2,-1,0], opening angle 50 deg)

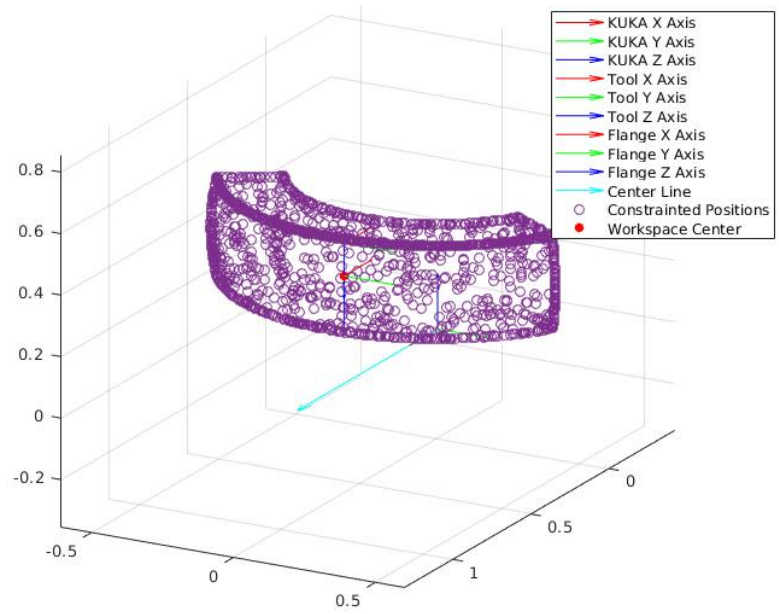

Figure 2.9: Workspace plot (center line [1,0,0], opening angle 120 deg)

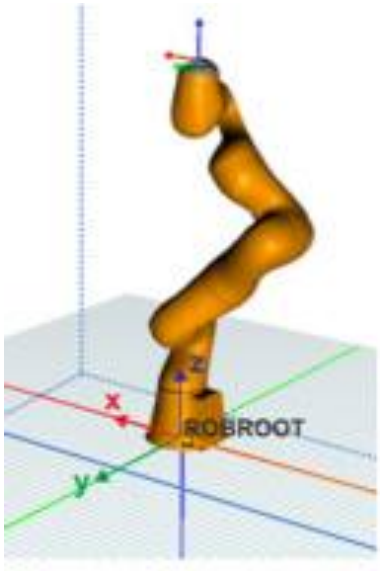

Figure 2.10: Coordinate systems [1]

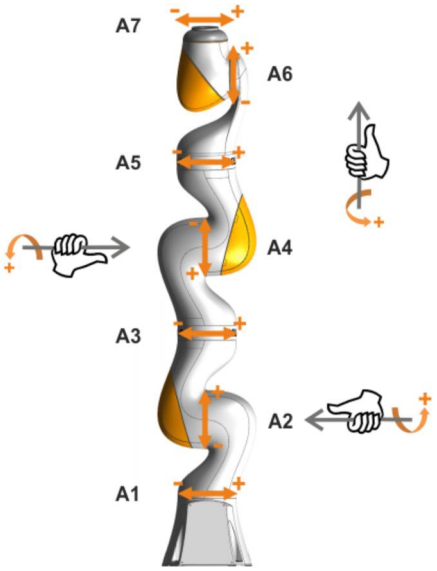

Figure 2.11: Joint convention [1]

2.1.7 Enforcing Workspace Boundary

During execution of the control scripts the `getConstrainedPosition.m` function is used to limit the position of the tool. There are four main parts to the enforcement of the workspace boundary.

1. If the desired tool position is above/below the allowed Z limits the constrained position is projected down/up along the shortest path to the allowed Z limit (shortest path is always along z axis). The range of motion in the XY plane is not restricted unless no other boundary constraint is enforced.
2. If the desired tool position is further from/closer to the origin of joint 2 than the allowed spherical limits the position is projected back to a valid position along a vector pointing perpendicular from the z axis to the desired position.
3. If the desired tool position is beyond the planes which are defined by the center line and the opening angle, the position will be projected down to the closest valid point on the workspace boundary (along normal of these planes).
4. If the desired tool position lies within any of the red areas in figures 2.6 and 2.7 (i.e. in negative center line direction or in negative z direction) an error is thrown. This is a safety precaution as positions that lie within these areas are unfeasible and might potentially be dangerous. If the control scripts recognise such an error the application is shutdown.

2.1.8 Home Position Calculation

In this software the home pose is always the idle position that the robot reaches when the application is started. The zero position is the position in which the arm is fully stretched out (i.e. joint positions [0, 0, 0, 0, 0, 0, 0]). This position is reached when the script successfully shuts down.

Note that the home position is not calculated if the parameter `use_home_pos` equals to true. If it equals to true the system will take whatever home position is specified by the `home_pos` parameter. The given home position must lead to a tool position that lies within the defined workspace. The workspace center parameter is set to the cartesian tool home position (as the software scales around this point). The workspace center is uploaded to the ROS parameter server by the control script and hence becomes visible to other nodes.

If the `use_home_pos` parameter equals to false, the home position will be calculated from the workspace definition and corresponding to the below constraints.

1. Tool position must coincide with workspace center
2. Tool Z axis must point downwards
3. Tool X axis must align with workspace center line
4. All joints must lie in plane spanned by center line and KUKA Z axis
5. Joints 3, 5 and 7 are fixed (i.e. 0)

The following paragraphs describe how the workspace is calculated if the `use_home_pos` parameter equals to false. Firstly, the workspace center is calculated like Figure 2.12 shows. The function `getJointsHome.m` computes the home pose from the workspace center and the above constraints. It simplifies the inverse kinematics problem to a 2R planar robot. Joint position 1 is calculated such that the rotation axis of joint 2 is perpendicular to the simplification plane in Figure 2.13. The simplification plane is spanned by the specified center line and the KUKA Z axis. Joint position 2 and 4 are calculated such that joint 6 lies in its correct position (above workspace center). The elbow up solution is chosen. Joint 6 is calculated such that tool Z axis faces downwards.

The script throws an error when the given workspace parameters would lead to a non-reachable home pose. A reason for this might be that the tool length that was specified is too long.

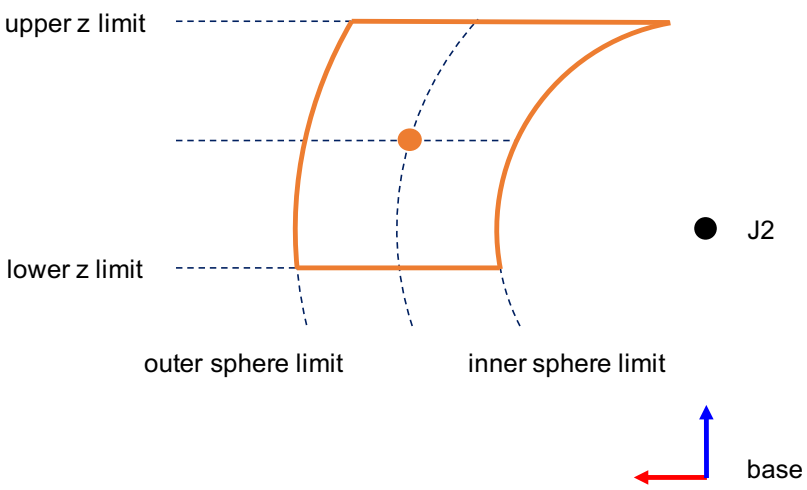

Figure 2.12: Calculation of workspace center

Since a 360 degree variability is required but the limits of joint 1 are +/- 170 degrees two cases are distinguished (see Figure 2.13). If the center line points in the positive X direction (including [0,1,0] and [0,-1,0]) the home position will resemble the left picture (also see Figure 2.14). If the center line points in the negative X direction the home position will resemble the right illustration (also see Figure 2.15). Depending on which home pose is initially calculated (tool X axis points towards or away from base frame) the operator needs to publish the correct desired tool frame transform to the system.

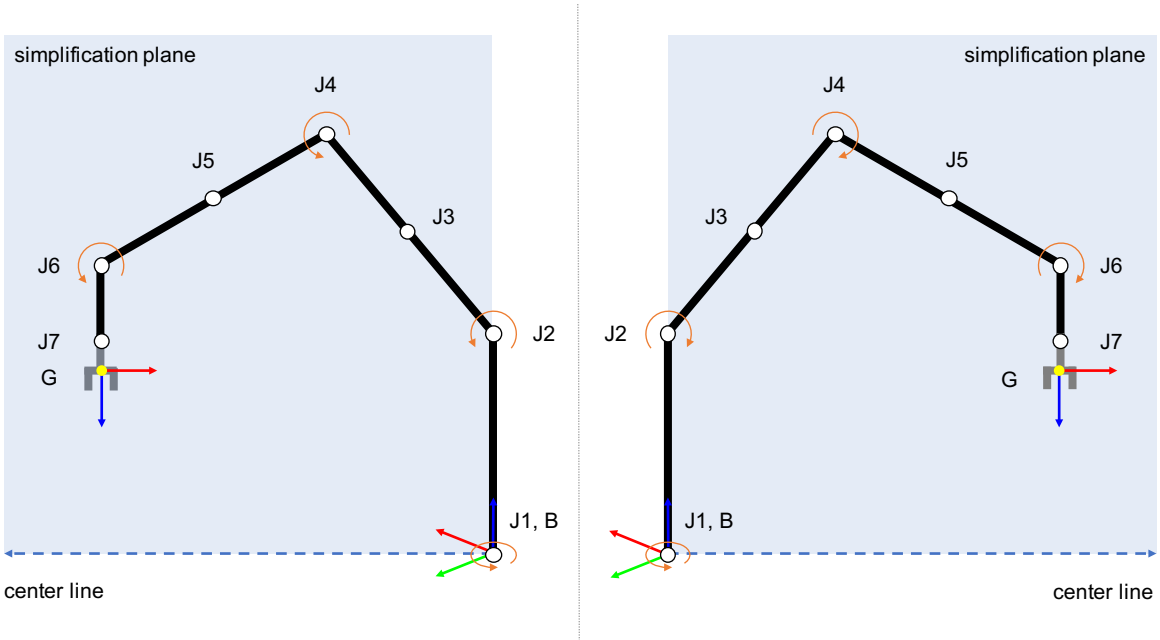

Figure 2.13: Calculation of home position

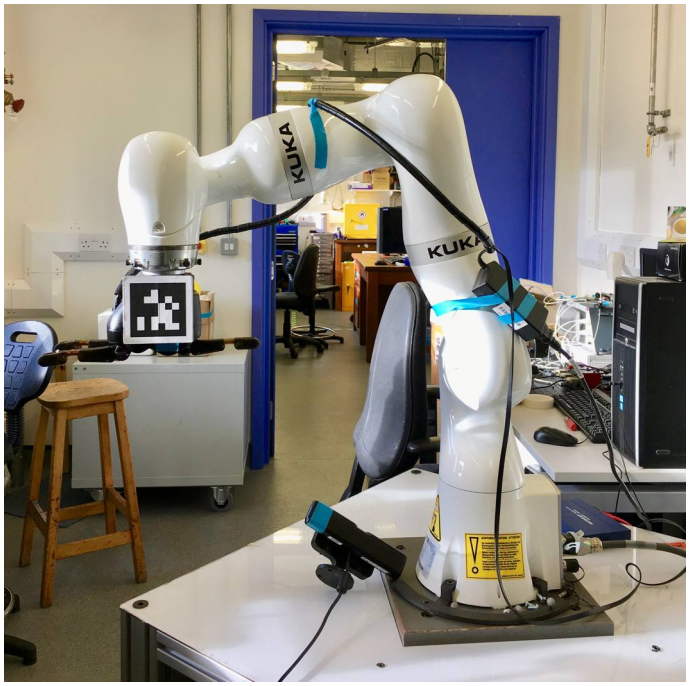

Figure 2.14: Home position with center line [1,0,0]

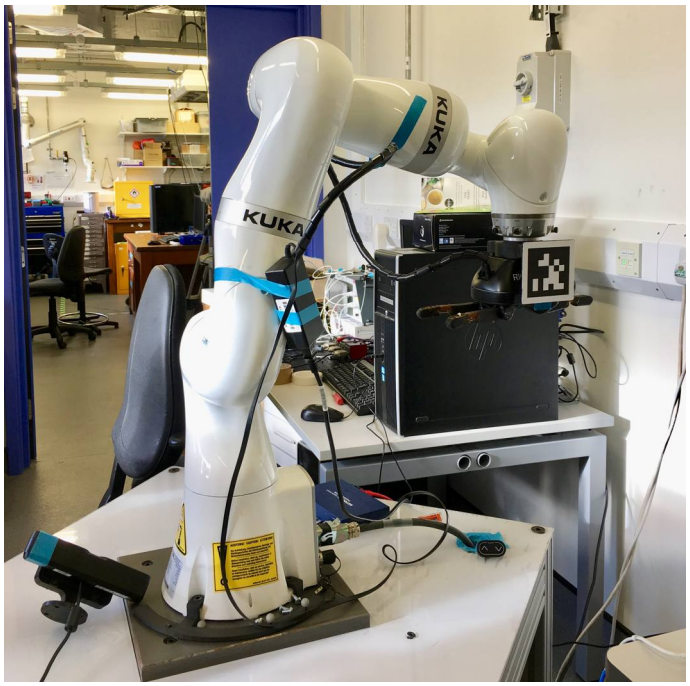

Figure 2.15: Home position with center line [-1,0,0]

2.1.9 Custom Home Pose

Seeing that depending on the application (and mounting position of the KUKA) some users may prefer a custom home pose as opposed to the calculated one, the KUKA control nodes offer an option to specify a custom home position in joint space. Simply enter the desired custom home pose in the configuration file under the parameter `home_pos`. To use this home pose the `use_home_pos` parameter must be set to `true`. Otherwise the robot will calculate the home pose as described in subsection 2.1.8. As described earlier the reference point for scaling in the control scripts is the workspace center. Again, the workspace center parameter is set to the tool tip in its home pose.

To find a home position that is suitable for your task consider moving the robot in the desired position using the KUKA touch pad and then simply transferring the displayed joint values to the config file. Please note that it is the operators responsibility to ensure that the tool position in the specified home pose is within the given workspace boundary. Consider adjusting the workspace boundary to fit your custom home position. The script will throw an error if you fail to set a home position inside the boundary. Again, for visualisation the Matlab script `workspacePlotter.m` can be helpful. See Figure 2.16 for an example (note the tilted tool transform and the workspace center).

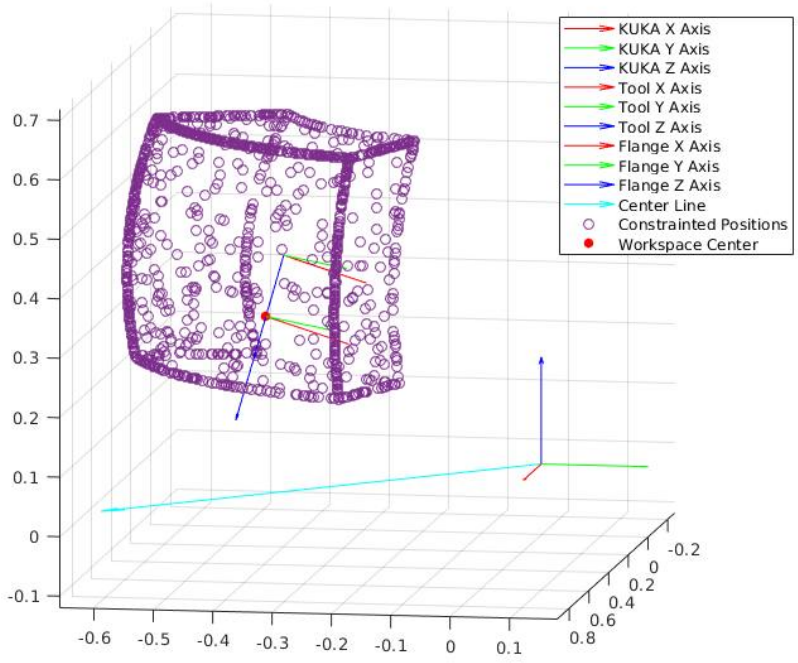

Figure 2.16: Plot of workspace custom home position [-45, 20, 0, -100, 0, 40, 0]

2.1.10 Control Loops

Algorithm 1: PTP control loop

```
1  while true do
2      if exit_app_emergency then
3          return;
4      else if exit_app then
5          break;
6      if control active then
7          if time out and not at home then
8              PTP motion to home position;
9          else
10             get tool transform;
11             scale position;
12             enforce workspace boundary constraints;
13             solve inverse kinematics;
14             check solution validity;
15             if movement above threshold then
16                 PTP motion to calculated joint position;
17     else if control inactive and not at home then
18         PTP motion to home position;
19     publish robot state;
```

In line 15 of Algorithm 1 it is checked whether the individual joint movements to the received pose are bigger than a certain threshold (can be set in the configuration file under parameter `joints_thresh`). The idea behind this is that if the ROS node receives a noisy transform signal the robot does not do PTP movements back and forth unless a threshold is surpassed. If every new transform message should be executed the parameter should be set to 0.

Algorithm 2: Real-time control loop

```
1  while true do
2      if exit_app_emergency then
3          return;
4      else if exit_app then
5          break;
6      if control active then
7          if time out and not at home then
8              PTP motion to home position;
9          else
10             get tool transform;
11             scale position;
12             enforce workspace boundary constraints;
13             solve inverse kinematics;
14             check solution validity;
15             if not mirroring and at home then
16                 if big jump then
17                     PTP motion to pick-up pose;
18                 else
19                     start real-time control directly;
20             else if mirroring then
21                 real-time motion to calculated joint position
22     else if control inactive and not at home then
23         PTP motion to home position;
24     publish robot state;
```

Line 16 of algorithm 2 checks whether the movement from the home position to the pick-up pose (i.e. the first tool transform received) surpasses a certain threshold (set to 2 degrees for each joint in the Matlab code). If the threshold is surpassed (e.g. because the input device is not well calibrated with the home tool pose), then a PTP movement is done to the pick-up pose, to avoid an abrupt start. Otherwise the node enters real-time mode directly.

In both scripts the timeout is adjustable in the configuration file under parameter `time_out`. If no new transformation message is published to the script for longer than the time out allows, the robot will move to its home position. While the time out is not yet expired, it will stay in the pose which was last published to the ROS network.

2.1.11 Running Control Scripts

This is the standard workflow to use the ROS control nodes, written in an instructional format.

1. Update configuration file (`kuka_params_rt.yaml` or `kuka_params_ptp.yaml`) to your needs. Consider visualising the workspace definition first using the `workspacePlotter.m`.
2. Make sure the defined workspace is clear and no obstacles are within it.
3. Boot the robot, establish ethernet connection and start MatlabToolboxServer application on KUKA robot controller in automatic mode.
4. Upload workspace configuration parameters to ROS parameter server. You can use the included launch files named `kuka_params_rt.launch` or `kuka_params_ptp.launch` for this, depending on which workspace you want to upload.
5. Run Matlab nodes for the control method you need (either `realTimeController.m` or `ptpController.m`).
6. Robot now moves to calculated (or otherwise specified) home pose.
7. Activate the control mode via a ROS service call to `kuka/kuka_control` or via the GUI.
8. Robot starts to listen to the `tf/` topic.
9. Make sure you publish the right tool transformation depending on if the tool's X axis points towards or away from the base. Refer to Figure 2.13 for more information on this.
10. Publish on the `tf/` topic. You need to provide the transform from `kuka/base_frame` to `kuka/flange_frame/desired`. Make sure every message contains an updated time tag.
11. Robot moves to specified tool transforms.
12. If you activate the gripper control (using the `kuka/gripper_control` service or the GUI) a corresponding boolean flag is published on the `kuka/gripper_control` topic.
13. Switch the KUKA control off with the service `kuka/kuka_control` or the GUI when you do not need it anymore. The robot moves home. You can reactivate it anytime.
14. When done, exit the application with the `kuka/exit_app` service or the GUI. Robot moves to zero position and terminates connection to KUKA robot.

You should pay attention to these things while using the ROS KST interface.

- If you are in real-time mode and the first transform you send is far away from the home position (in joint space) the robot will perform a PTP movement to the pick-up position. From this point onwards all further transforms are executed in real-time mode. This should avoid abrupt starts if the initial tool transform you send does not lie in the close vicinity of the home configuration (e.g. due to miscalibration). However, if the first transform you send is sufficiently close to the tool home transform the robot will execute in right real-time mode straight away.
- For the real-time control mode a **frame rate of at least 50 Hz** is recommended. For the PTP mode the frame rate can be lower. However, make sure that the frame rate is high enough such that timeout threshold is not reached. If the Matlab ROS node does not receive any new `tf/` messages for longer than the threshold allows, it will move to its home position.
- This is especially relevant in real-time mode: it is your responsibility to do **adequate filtering** of the transform message before sending the transforms to the control node. A sequence of good transform messages for the real-time node should form a smooth trajectory. For example, this can be achieved by simple averaging filters or more complex Kalman filters. The KUKA control node simply executes the received transform messages and does not filter.
- If in your task it is cumbersome to Figure out the full transformation from the KUKA base frame to the desired tool frame you could control the robot relative to its home pose. The home pose of both the flange and the tool in the base frame are constantly published to the ROS tf tree. These fixed transforms are called `kuka/flange_frame/home` and `kuka/tool_frame/home` respectively.
- The `ptpController.m` always moves with the relative velocity specified in the parameter `velocity_ptp` of the PTP configuration file.
- The `realTimeController.m` moves to its home position (beginning) and zero position (end) with `velocity_ptp_slow`. All PTP movements in the meantime (i.e. back to home position when no more new data is incoming or when KUKA control stopped) are executed with relative velocity `velocity_ptp_fast`.
- In both the real-time mode and the PTP mode, the ROS node will disregard any incoming transform messages that are sent while the robot is executing a motion. It will only listen to the most recent transform message on the tf tree, once its previous movement was executed.
- All movements are scaled around the tool tip in the home position if your scaling factor does not equal to 1. Remember: using the calculated home pose, this corresponds to the geometrical workspace center (see Figure 2.12). Using a custom home pose the movement will be scaled around the tool position in the given home position.

- If you wish to deactivate all workspace constraints in a control script look for the function call of `getConstrainedPosition` and delete it. Note: deactivating the workspace boundary might be dangerous. The robot can move freely and will execute any transform you send unless it is not executable by the robot.

2.2 Leap Motion

2.2.1 Leap Configuration File

The configuration file is called `leap_params.yaml` and is part of the `leap_kuka` ROS package.

Paramater Name	**Default**	**Note**
z_clearance	0.15	remaining hand offset when lower z limit reached
box_dimensions	[0.2, 0.2, 0.2]	dimensions of interaction box
buffer_size	10	size of averaging buffer
start_tol_pitch	20	tolerance for hand pitch at pick-up
start_tol_roll	20	tolerance for hand roll at pick-up
run_tol_yaw	25	tolerance for hand yaw while running
run_tol_pitch	25	tolerance for hand pitch while running
run_tol_roll	25	tolerance for hand roll while running
min_confidence	0.05	do not use data if below minimum confidence

Table 2.5: Leap Motion configuration parameters

Table **??** shows a list of all transformations published by the `leap_kuka` node. The following paragraphs explain how each of the transformations is obtained.

From	To
kuka/base_frame	leap_motion/virtual_frame
leap_motion/virtual_frame	hand/frame
leap_motion/virtual_frame	kuka/tool_frame/desired
leap_motion/virtual_frame	kuka/flange_frame/desired

Table 2.6: Transformations published by leap_kuka node

leap_motion/virtual_frame

The calculation of the transform from KUKA base to virtual Leap Motion relies on two ideas. The position of the Leap Motion lies directly below the workspace center (i.e. tool home position). The Z offset from the workspace center is the distance between workspace center and lower Z limit plus the Leap Motion clearance parameter. A graphical overview of this is given in Figure **??**. As the workspace center must be known before the virtual Leap Motion position can be calculated it must be uploaded to the ROS parameter server before launching the Leap KUKA node. This is done by the control scripts presented earlier.

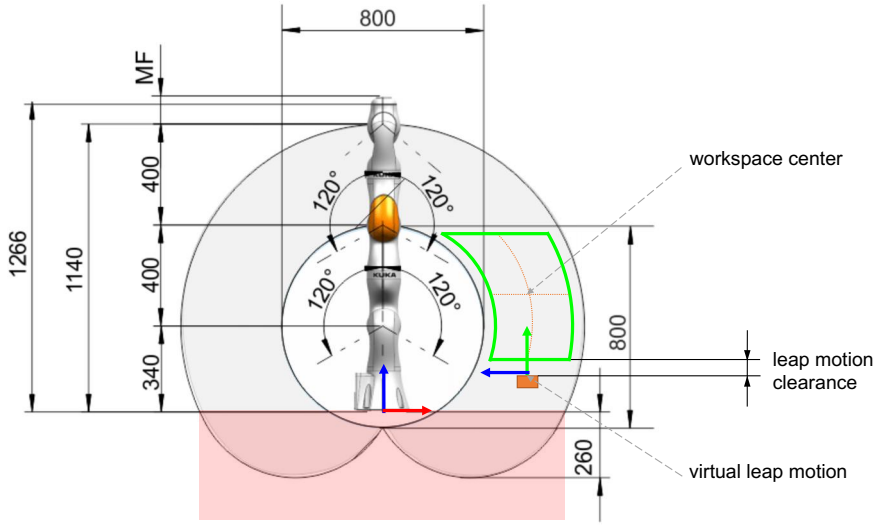

Figure 2.21: Position of Leap Motion in KUKA base frame

Furthermore, the orientation of the Leap Motion is determined by two constraints. Firstly, the Leap Motion Y axis must be parallel to the KUKA Z axis. Secondly, the Leap Motion must be rotated around its Y axis such that the XY components of the first detected hand's direction vector points in same direction as the center line. This means that the hand frame measured by the Leap

Motion and the fixed hand frame in Figure **??** must line up. Therefore it is completely arbitrary with which yaw angle the user's hand approaches as the Leap Motion will always be rotated around its Y axis to fulfil the second constraint. Please note that users should actually align their hands with the robots center line when using the system. Otherwise the control of the robotic rig might feel unintuitive.

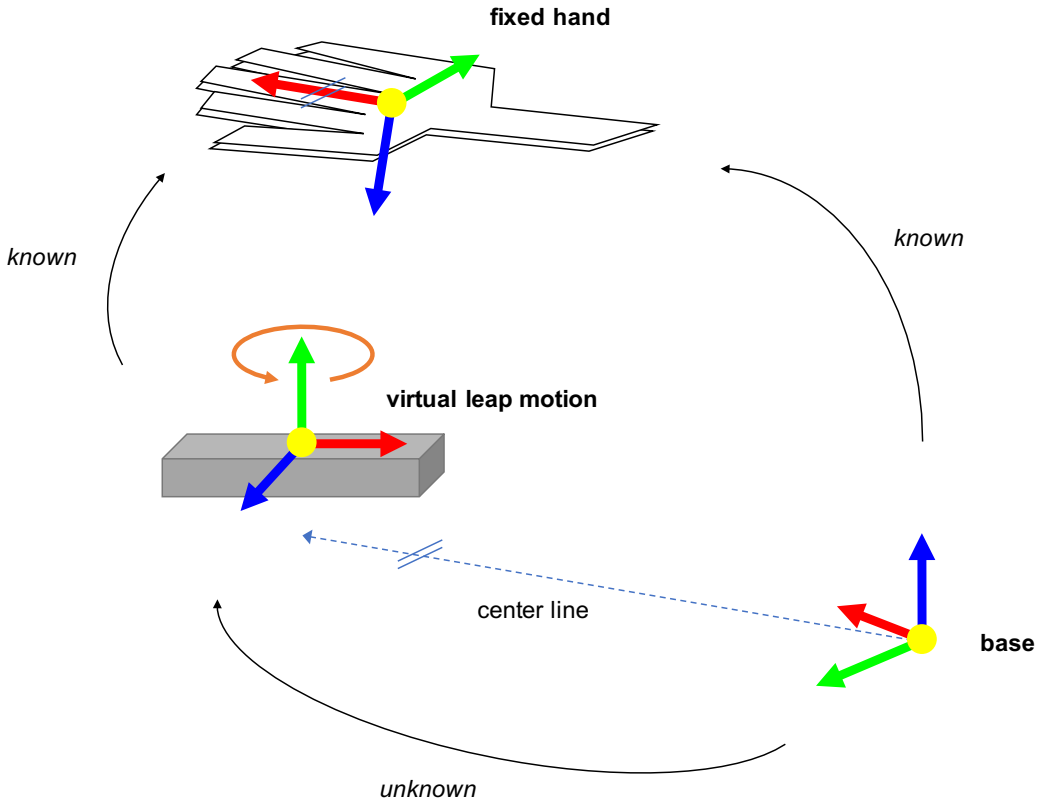

Figure 2.22: Transformation to virtual Leap Motion

hand/frame

The hand frame is simply the transform determined from data which the Leap Motion collects about the hand. The transform is computed from the hand's X, Y and Z axes (refer to Figure **??**) and the palm's center position.

kuka/tool_frame/desired

As mentioned earlier the measured hand frame should be mapped to the desired tool frame. The position of the desired tool frame always equals the position of the measured hand.

The orientation includes a further step. Since the measured hand might exceed the running tolerances specified in the Leap Motion configuration file an adjusted hand frame is introduced (which is not published to the tf tree). The adjusted hand frame equals the measured hand frame if yaw pitch and roll are within the running tolerances. However, if the measured hand exceeds the allowed yaw, pitch or roll the hand's orientation must be adjusted as depicted in Figure **??**. Yaw, pitch and roll are measured around the fixed hand frame. If any of the yaw, pitch or roll constraints are exceeded the adjusted hand's rotation is simply the measured hand's rotation with adjusted values (which if exceed correspond to the maximum allowed rotation and otherwise represent the measured hand's rotation).

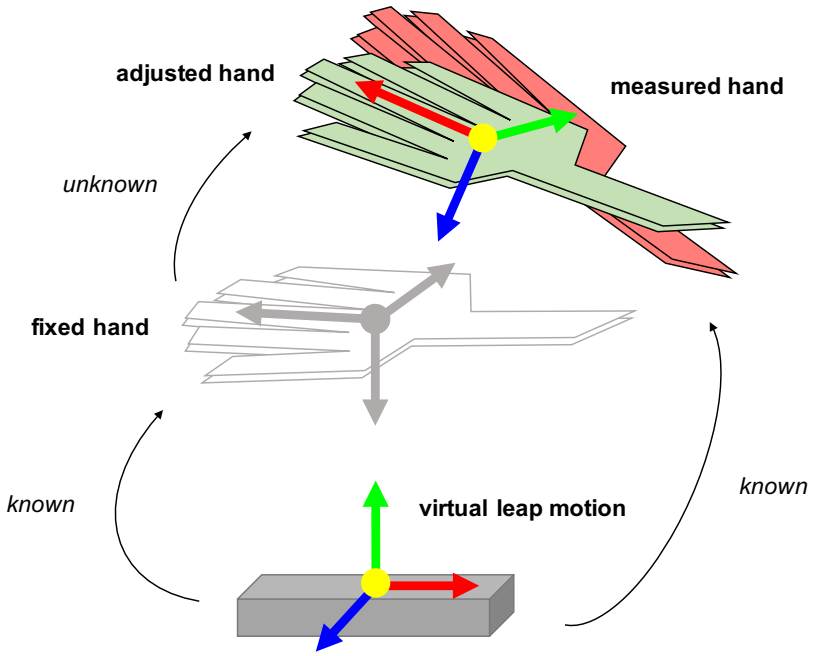

Figure 2.23: Transformation to desired KUKA tool pose

It is assumed that the automatic home pose calculation feature from section **??** is used. Depending on where the center line points the home tool orientation with respect to the base can look different (refer to Figure **??**). Depending on where the center line points the adjusted hand's frame is rotated 180 degrees around its Z axis and then published to the tf tree.

kuka/flange_frame/desired

The desired flange frame is simply the tool frame translated by the specified tool length along its negative z axis. It is not used to control the KUKA robot, but still published for completeness.

Figure **??** shows a final overview of the frames published by the Leap KUKA node (except the flange frame for better visibility). Note how the rotation of the `hand/frame` exceeds one of the rotational constraints and the `kuka/tool_frame/desired` frame is rotated less. The position of both frames coincides.

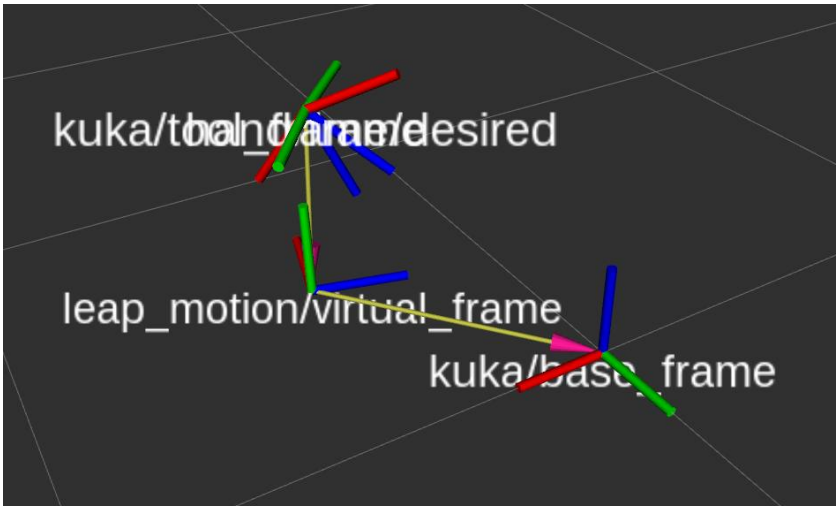

Figure 2.24: Frames published by leap_kuka node viewed in RViz

Things users should pay attention to when using the `leap_kuka` ROS node.

- To start publishing the desired tool frame put a hand inside a virtual interaction cube above the Leap Motion. Keep hand horizontal (within specified tolerances). Leap Motion orientation on table does not matter. Node uses a FIFO queue for averaging of data. Once enough data for the averaging process is collected node publishes the desired tool transform.
- The system does not distinguish between left or right hands. You can use either hand.
- If two hands are detected the system always regards the hand which has been visible the longest.
- Switch "Auto-orient Tracking" in LeapControlPanel settings on to take leverage of full capability of ROS node. If switched off, the Leap Motion only recognizes hands coming in from one lateral side.
- If for debugging purposes (never for control!) you would like to run the Leap KUKA node independently you can do so using the `leap_kuka_debug.launch` launch file. Note that in order to make the node run independently this file uploads default values for the `center_line` and `ws_center` arguments - hence only use it for debugging and visualisation.

3 Robotic Hand Node

3.1 General

The ReFlex Takktile robotic hand by RightHand Robotics is a 4 degree of freedom force sensitive robotic gripper. It can be conveniently controlled through the ROS driver which is included in the `modules` folder. Before attempting to control the robotic hand make sure that the networking setup (see section **??**) was completed successfully. Below is the workflow for controlling the robotic hand.

1. Power robotic hand.
2. Establish valid ethernet connection.
3. Type `roslaunch reflex reflex_takktile.launch` in a terminal to launch the robotic hand's ROS driver.
4. Calibrate the hand before using it (especially before first usage) by calling the services `reflex_takktile/calibrate_fingers` and `reflex_takktile/calibrate_tactile`. The calibration results will be stored in a directory inside the hand ROS driver. Make sure the hand is in an adequate zero position (i.e. fingers stretched out) before calibrating.
5. Publish to the right closure angles to the topic `reflex_takktile/radian_hand_command`. The message must be of type RadianServoCommands and consists of four numbers in the order [index, middle, thumb, spread]. The opened hand command corresponds to [0, 0, 0, 0] and a closed hand command would correspond to [2, 2, 1.8, 0].

3.2 Leap Motion

The Python script `leap_hand.py` forms a ROS node to control the robotic hand intuitively using the Leap Motion gesture tracking device. The control node calculates four angles from vectors that it constructs starting from the palm center position. In Figure **??** alpha represents the index finger flexion, beta the middle finger flexion and gamma the thumb flexion angle. Angle delta (not in figure) represents the spread between index and middle finger.

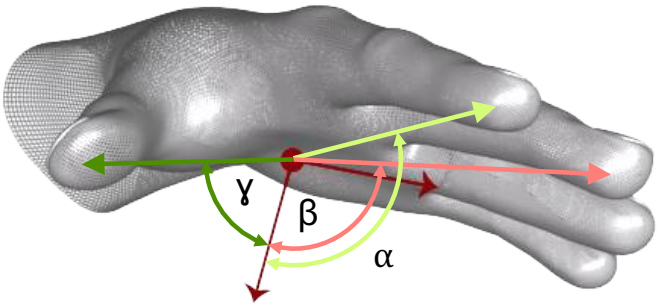

Figure 3.1: Human flexion angles [**LeapHand**]

1. **alpha**: angle between palm normal and vector to index finger tip
2. **beta**: angle between palm normal and vector to middle finger tip
3. **gamma**: angle between palm normal and vector to thumb tip
4. **delta**: angle between vectors from palm center to center of intermediate bones

Once the `leap_hand.py` node calculated the human flexion angles it maps the measured angles to the minimum and maximum specified flexion angles of the robotic hand. This allows for smooth opening of the individual fingers (as opposed to simple open/close movements). If no hand is detected the robotic hand flexion equals is set to the open pose. The array of robotic hand flexion angles is published to the `reflex_takktile/radian_hand_command` topic. The script works with either hand.

4 Tips and Tricks

4.1 Leap Rig Demo

These are short instructions for users that would like to show the demo of Leap Motion control of the robotic hand and the KUKA robot. These instructions may repeat information previously presented in the report.

1. Adjust configuration files `kuka_params_rt.yaml` and `leap_params.yaml` to your needs (especially consider reading through subsection **??** on the workspace definition).
2. Setup system as shown in Figure **??** (robot, hand and computer connected via ethernet through a switch and Leap Motion connected to computer using USB)
3. Start KUKA Robot with green toggle on robot controller. Wait for it to boot.
4. On the KUKA touch pad, press "Applications" in the top left and click on the application "MatlabToolboxServer".
5. Mount and power the Robotic Hand. Make sure that its fingers are in a flat (i.e. opened) position as this will be its zero position.
6. Establish network connection to KUKA and robotic hand by clicking on the devices in "Ethernet Networks" in the top right corner. If you click on "kuka" or "hand" it should say "Connection Established".
7. Make sure the defined robot workspace is clear.
8. Run the application on the KUKA touch panel by clicking green arrow pointing to the right (button is on the left of the touch pad). Make sure that you are in automatic mode. This can be changed by twisting the black knob in the top of the touch panel in a horizontal position. Change the mode to AUT if necessary and turn it back.
9. Navigate to the `leap_rig/shell` folder and launch both shell scripts in listing **??**. Note that if your workspace is not called `catkin_ws` you must adjust the path to your workspace name in the `run_leap_control.sh` script.
10. Once the robot moved to its home position, activate KUKA control and robotic hand control by ROS service calls or with the "ROS Control" toggles in the Matlab GUI.

11. Put hands inside a virtual interaction cube which sits above the Leap Motion. Keep hand horizontal. Leap Motion orientation on table does not matter. Once enough data for the averaging process is collected the robot will start mirroring the hand motions.
12. Switch the KUKA control off with the service `kuka/kuka_control` or the GUI when you do not need it anymore. The robot moves home. You can reactivate it anytime.
13. When done, exit the application with the `kuka/exit_app` service or the GUI. Robot moves to zero position and terminates connection to KUKA robot.

Listing 4.1: Launching Leap Rig demo

```
cd ~/catkin_ws/src/leap_rig/shell
./run_leapd.sh
./run_leap_control.sh
```

Please consider this when running the demo.

- Consider running the Diagnostic Visualiser when clicking on the LeapControlPanel (green rectangle in top left corner). It gives a nice demo of the Leap Motion system to people who are not yet familiar with it.
- When someone is controlling the robot always have hand on the KUKA emergency stop button.
- Should you leave the robot unsupervised press the emergency stop button or switch the robot off.
- You might want to increase the buffer size of the averaging queue of the Leap KUKA node in its configuration file. This will make the robot less responsive and lead to a smoother trajectory.
- Before switching the KUKA "ROS Control" on explain, that users have to put their hand inside a virtual cube above the Leap Motion to "pick up" the robot. Their hand has should be horizontal and still when put inside the interaction box. Further, their hand and forearm should be parallel to the robot arm to ensure intuitive control.
- Before letting others control the robot, let them get a feeling for the system by only letting them control the Robotic Hand first. Explain, that only the thumb, index and middle finger are used to control the hand. The other fingers should be as straight as possible. The reason for this is that if users make a fist, the Leap Motion will produce a noisy signal for hand position, direction and normal from which the tool transformation is calculated.

The following is a summary of safety features of the software setup.

- Robot performs a safety stop if it senses a cartesian torque of 25Nm or higher.
- The maximum end effector velocity is set to 800mm/s. Robot performs safety stop if this velocity is exceeded. The maximum torque and velocity are specified in the KUKA project which is synchronised to the controller (view KUKA Sunrise Workbench documentation).
- To start the Leap Motion control of the robot, the operator has to actively press the toggle "ROS Control". Only if this button is active, the system will pick up the data from the hand inside the cube and start mirroring (i.e., real-time control).
- To operate the robot users have to put their hands inside a virtual cube directly above the Leap Motion. This ensures that the system is starting tracking in an area that is well perceived by the Leap Motion and signal noise is kept to a minimum.
- At startup users have to put their hands in a horizontal position. Tolerances are specified in the Leap Motion configuration file. This ensures that the robot tool moves to a reasonable position and orientation in a controlled manner.
- Very fast hand movements are compensated for in averaging process.
- At all times during mirroring, the software limits yaw, pitch and roll to a specified value in the Leap Motion configuration file.
- Emergency button in Matlab GUI instantly stops script. Don't press unless in an emergency. Use "Exit Application" for safe exit and return to zero position.
- All positions that would be sent to the robot are checked. If they are not inside the workspace, they are clipped to the workspace boundary.
- If any position is "very" infeasible (i.e., negative KUKA z axis or in negative center line direction) the program throws an error and terminates.

4.2 Troubleshooting

- If the robot stops while mirroring your hand movements, you probably exceeded the maximum end effector velocity (800mm/s). Pressing the "run" button (green triangle pointing to the right) should continue the session. You might want to increase the averaging buffer size to make the robot's movements less responsive.
- "Error using `KST/net_establishConnection`": If this error shows in Matlab, check the ethernet cables and make sure that you are actually connected to the robot in "Ethernet Networks" in the top right corner. If you press "kuka", your computer should say "Connection Established". Also make sure the MatlabToolboxServer is running on the robot controller. It shuts down again after a while if no connection request is made.
- "No KRC connection possible": If this error message shows on the KUKA touch panel after booting, try rebooting the robot. Ideally, unplug the robots ethernet cable from the switch before booting. The error should then be resolved.
- "Could not attach tool to the flange": You get this error message from the KUKA Sunrise Toolbox if the previous connection to the robot was not successfully shut down (happens if "Emergency Stop" in Matlab is performed). Reboot the robot.
- Java Bind Exception: If you get this message on the robot panel, this means your previous connection to the robot did not terminate successfully. Reboot the robot.
- Robot does not start mirroring: Check if the toggle "ROS Control" in the KUKA Panel of the GUI is activated and your hand is in a horizontal position and inside the interaction box. Also make sure that the Leap daemon is running. You could use the Diagnostic Visualizer (right click on green rectangle, LeapControlPanel needs to be running) to check if the Leap Motion is actually detecting a hand. If it does not, but it should, try relaunching the Leap daemon.
- No Connection to robotic hand possible: Make sure the power cable is plugged into the Robotic Hand. Unfortunately it does not always sit in place firmly. As with the KUKA make sure that your ethernet cable is correctly plugged in and you have a valid connection under "Ethernet Networks".
- If robot detected resistant force: The robot will pause the application running on the KUKA controller. You should be able to press the green button and unpause the motion. If that does not work or the robot ran into more problems, you might need to restart it.
- Other unexpected behaviour: Check the command window in Matlab, you might find valuable information there.

4.3 Leap Motion Details

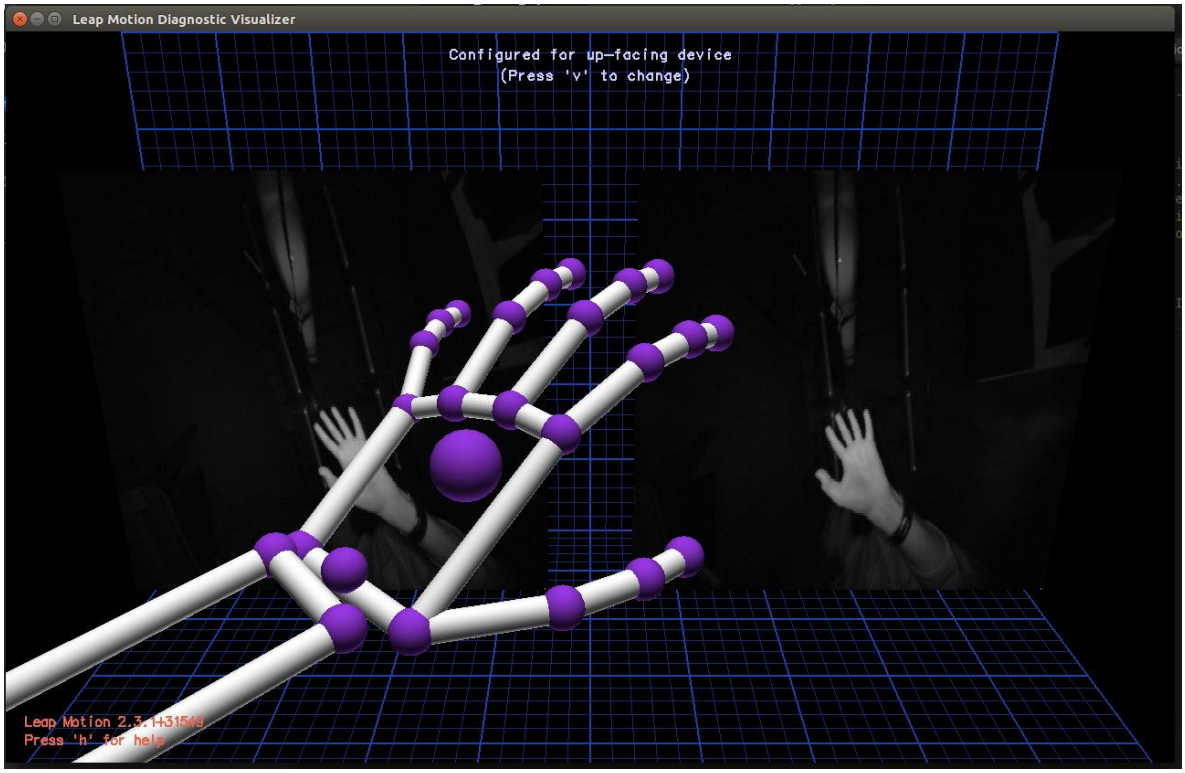

Figure 4.1: Leap Motion diagnostic visualizer

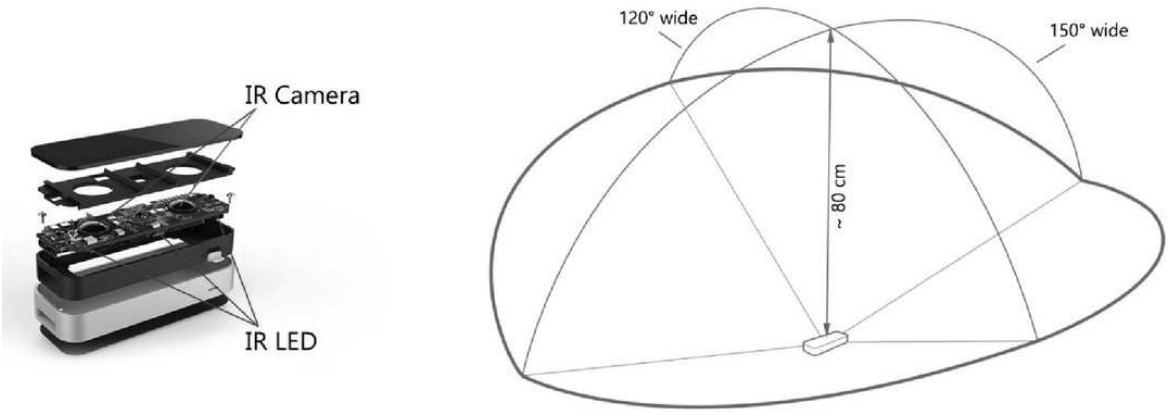

Figure 4.2: Leap Motion workspace [**Wozniak**]

4.4 ROS Commands

Listing 4.2: Useful ROS commands

```
rosclean purge
rosparam list
rosservice list
rostopic list
rosrun tf view_frames
rostopic echo topic_name
roscore
rqt_image_view
rosrun rviz rviz
catkin build
roslaunch pkg_name file_name.launch
```

5 Vision

5.1 Virtual Reality

To start VR support simply plug in the RGB-D camera and the Oculus. Make sure that the Oculus is detected as an external display by launching the nvidia-settings utility from the command line if necessary. The Ubuntu home screen should be displayed in the Oculus if correctly connected. Then simply launch the following script, which wires all drivers and topics up such that the video published by the camera will be displayed in the Oculus Rift. Figure **??** shows what is displayed inside the Oculus Rift. When the HMD is worn the two images blend together to form a single image.

Listing 5.1: Launching virtual reality node

```
roslaunch vision camera_oculus.launch
```

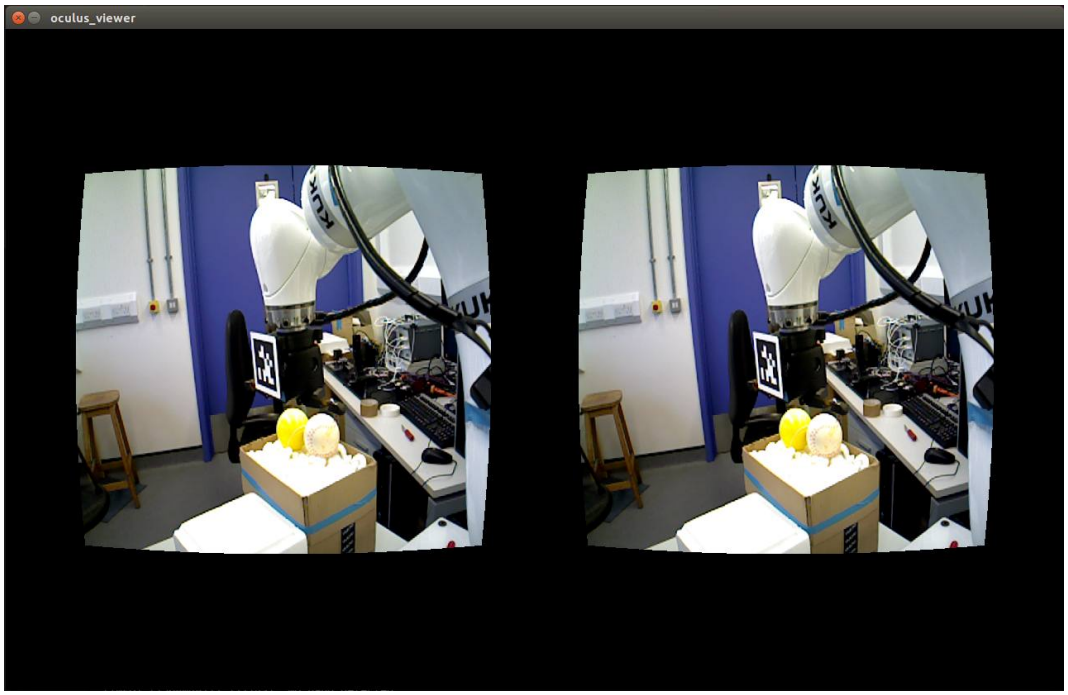

Figure 5.1: Workspace as viewed inside the Oculus

5.2 Visual Fiducial

Seeing that the long-term goal of this project is to investigate autonomous robotic grasping a reference between the RGB-D camera and the robot must be established. A first attempt at this is done using an AprilTag fiducial on a 3D printed marker panel. The camera recognises a predefined AprilTag of known size and calculates its position and orientation from the camera intrinsics. Whether this is sufficiently accurate for the purposes of autonomous robotic grasping still needs to be investigated. Note that the AprilTag does not always need to be visible if the calibration is done before starting the task. This is due to the fact that the transform from KUKA base to camera does not change as both the camera and the KUKA are static in this scenario. Also note how in Figure **??** the `tag_frame` exist in the tree but is currently not displayed.

Listing 5.2: Launching camera AprilTag node

```
roslaunch vision camera_apriltag.launch
```

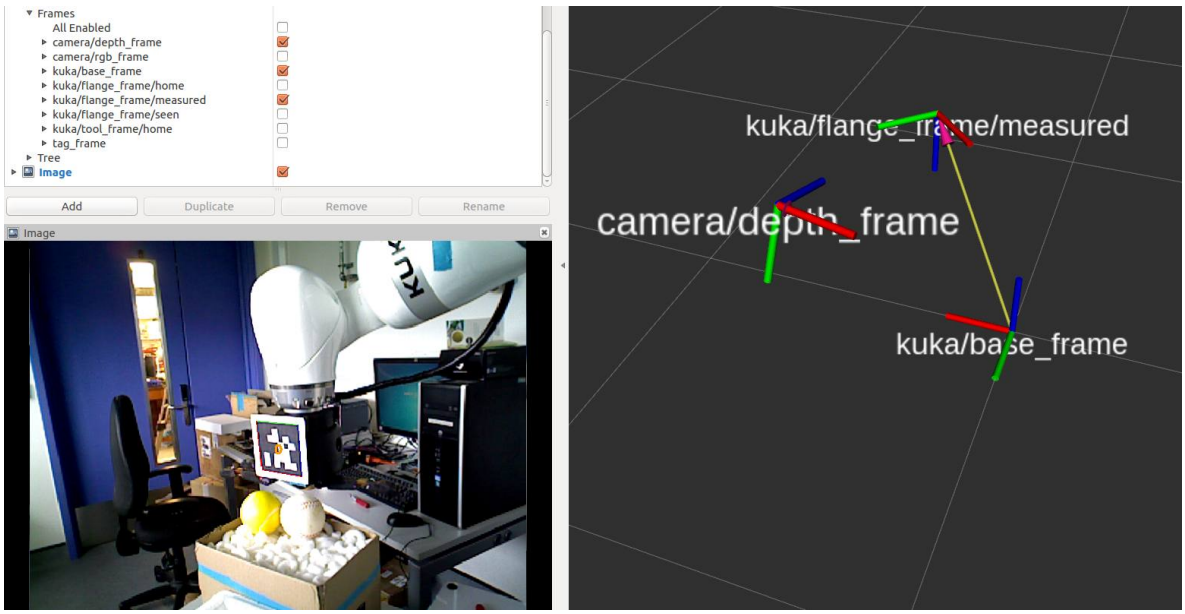

Figure 5.2: Camera pose calculation in KUKA base frame

"AprilTag is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration. Targets can be created from an ordinary printer, and the AprilTag detection software computes the precise 3D position, orientation, and identity of the tags relative to the camera. The AprilTag library is implemented in C with no external dependencies. It is designed to be easily included in other applications, as well as be portable to embedded devices. Real-time performance can be achieved even on cell-phone grade processors. The fiducial design and coding system are based on a near-optimal lexicographic coding system, and the detection software is robust to lighting conditions and view angle." [**AprilTag**]

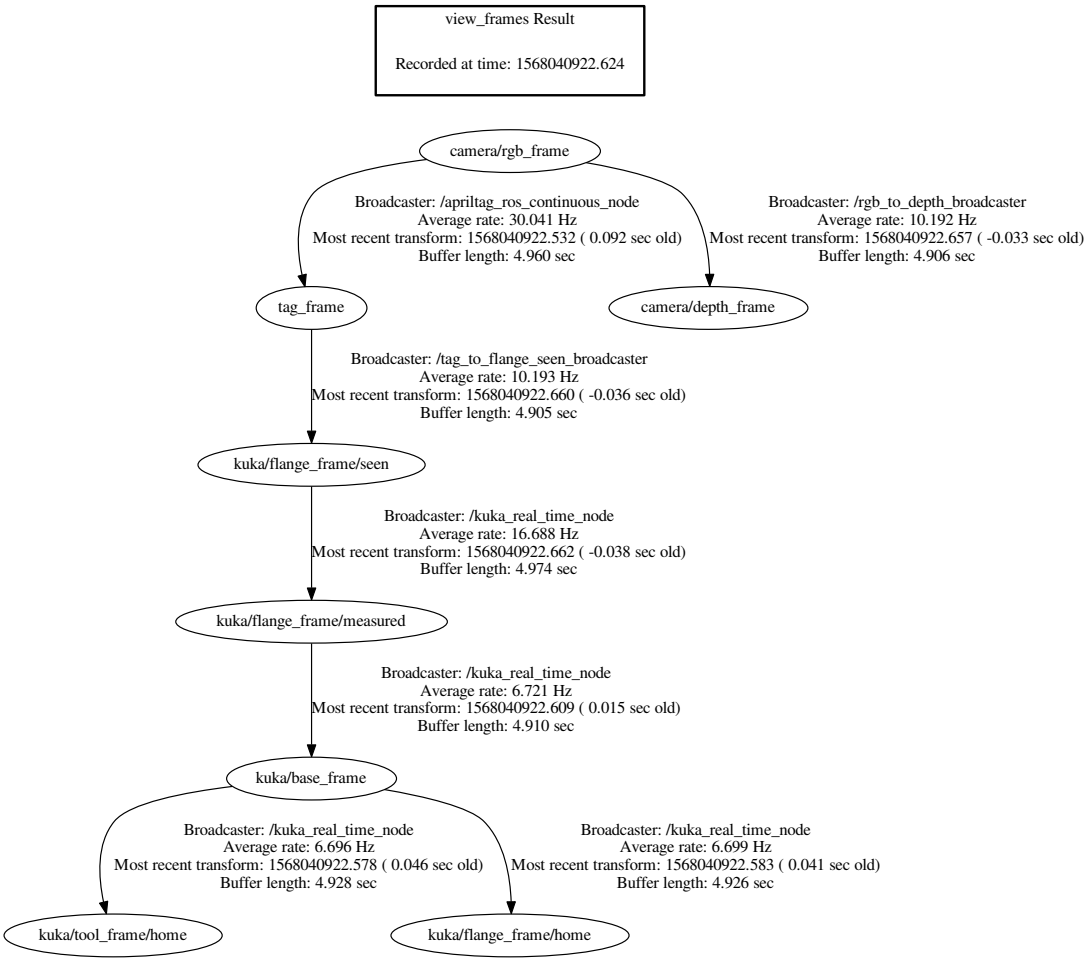

Figure 5.3: Tf tree when camera_apriltag and a KUKA control node running

5.3 RGB-D Fusion

To merge the camera's RGB data with its depth data a launch file was set up in the `vision` folder. It uses the `image_pipeline` module. To launch the RGB-D fusion type in the following. A preconfigured RViz window opens (see Figure **??**) to inspect the fused data.

Listing 5.3: Launching RGB-D fusion nodes

```
roslaunch vision register_rgbd.launch
```

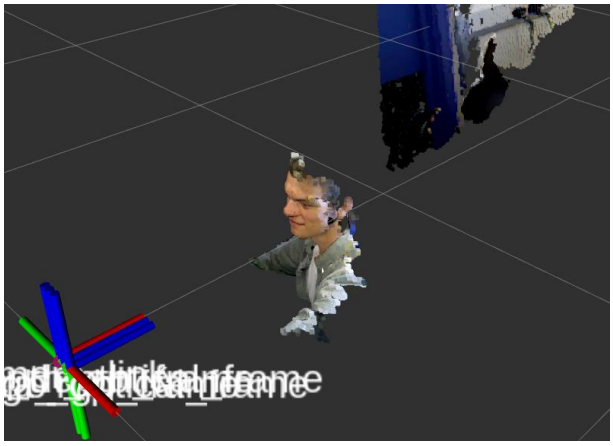

Figure 5.4: RViz showing fused RGB-D data

For autonomous grasping prediction using depth data (see chapter **??**) a reference between the KUKA base frame and the camera depth frame must be established. The approach used for this estimation in this project currently is determining the cameras RGB optical frame using the AprilTag library and then rigidly transforming to the depth optical frame of the camera. The merged RGB-D data produced by this launch file could be used to get a more accurate estimate of the tag transform relative to the camera depth frame. This might be a sensible approach:

1. Detect AprilTag using RGB camera
2. Get RGB pixels that correspond to the marker panel
3. Fuse with depth pixels
4. Reconstruct the transform from AprilTag to depth frame directly from landmarks (e.g. translation is depth pixel's value of tag's center and orientation could be calculated by constructing a plane that intersects all four corners of tag)

It is worth investigating if higher accuracies can be achieved for the estimation of the depth frame using this approach compared to the current one (see chapter **??**).

6 Autonomous Grasping

This is a first attempt at investigating autonomous grasping using a RGB-D camera and neural networks to process depth images. The proposed research question is whether the performance of existing neural networks (such as presented in [**Grasping**]) can be enhanced by a human in the loop. Since an intuitive user interface for the KUKA robot and robotic hand has been developed in this project it should be investigated whether performance scores can be improved using data that was generated from the systems presented earlier in this report.

As a first step the pretrained neural network presented in [**Grasping**] is used to get a first impression of the accuracy of their system and potential areas of improvement. See the network architecture of their so called generative grasping convolutional neural network (GGCNN) in Figure **??**. Start the code for grasping prediction with the terminal commands below. In the image viewer select the desired prediction topic (e.g. `ggcnn/img/grasp`).

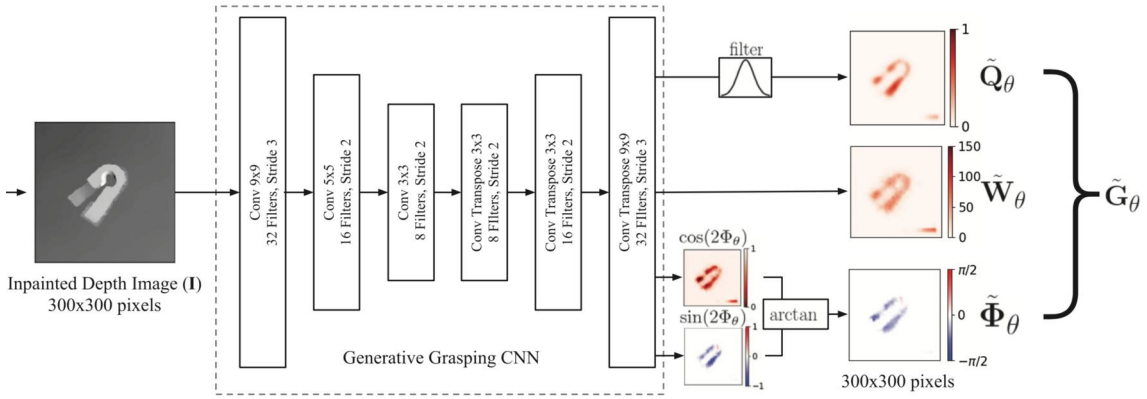

Figure 6.1: Network architecture of GGCNN [**Grasping**]

Listing 6.1: Launching grasp prediction node

```
# in a terminal type
roslaunch grasping grasping.launch
# in a new terminal type
conda activate grasping_py27 && rosrun grasping ggcnn.py
# when done
conda deactivate
```

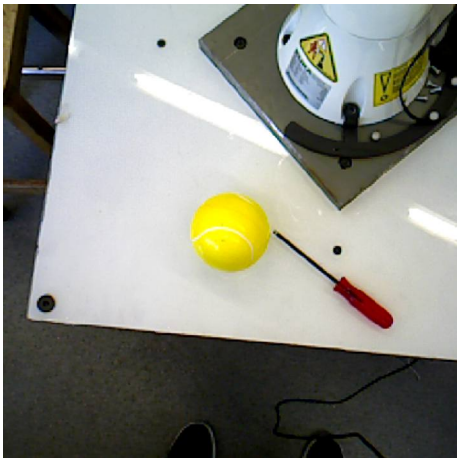

Figure 6.2: RGB view

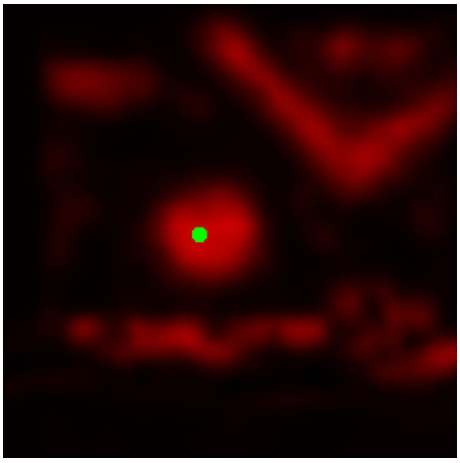

Figure 6.3: Grasping point prediction

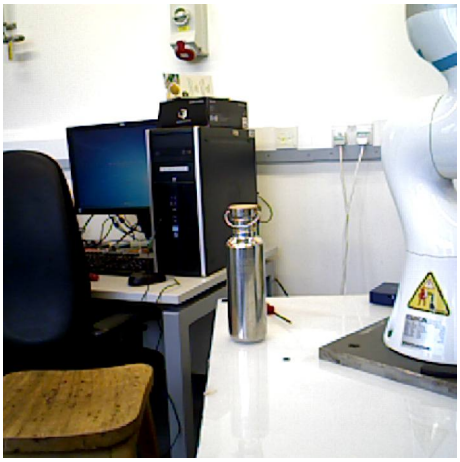

Figure 6.4: RGB view

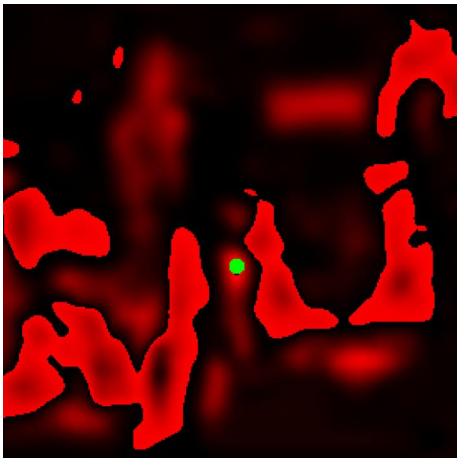

Figure 6.5: Grasping point prediction

The `TransformStamped` of the predicted grasping pose is also published (however with a fixed rotation). It still needs to be investigated how to best estimate the tool orientation from the neural network output (see chapter **??**). To see how Morrison et al. work with the predicted data and achieve closed loop control of grasping movements, it is recommended to check the code in the "GGCNN Kinova Grasping" GitHub repository.

Since the neural network's predictions are not very stable a post-processing filtering step should be considered. An additional neural network which is appended to the existing one (Figure **??**) could serve this purpose. It is most sensible to work with the whole grasping "heat map" that is produced by the first neural network (as opposed to just the final prediction) to not loose any valuable abstraction the GGCNN makes. A second network could then use this preprocessed data and predict the full transformation from camera depth frame to the grasping target. This transform can then be transferred into the KUKA reference frame with a relation that has been established with Hand-Eye calibration using the AprilTag fiducial.

Training the second neural network could be done using data that was collected using the Leap Motion KUKA node. A dataset of possible grasping poses could be recorded from desired tool transform which is continuously published by the Leap Motion KUKA node. If this is sampled at a high frequency and paired with the corresponding GGCNN prediction a dataset for training of the second network could be build up. The workflow for dataset collection could look like this.

1. Determine transform from KUKA base to camera depth frame
2. Choose one item to grasp (e.g. cup)
3. Place object in workspace and keep robot in home position
4. Keep camera on tripod and start GGCNN
5. Collect 1000 predicted heat maps of workspace (approx. 1 minute at 15 Hz prediction rate)
6. Use Leap KUKA node to generate 1000 possible grasping tool transforms (approx. 30 seconds at 30 Hz sampling rate)
7. Choose other objects and repeat steps 2 to 6
8. Train appended neural network with collected data

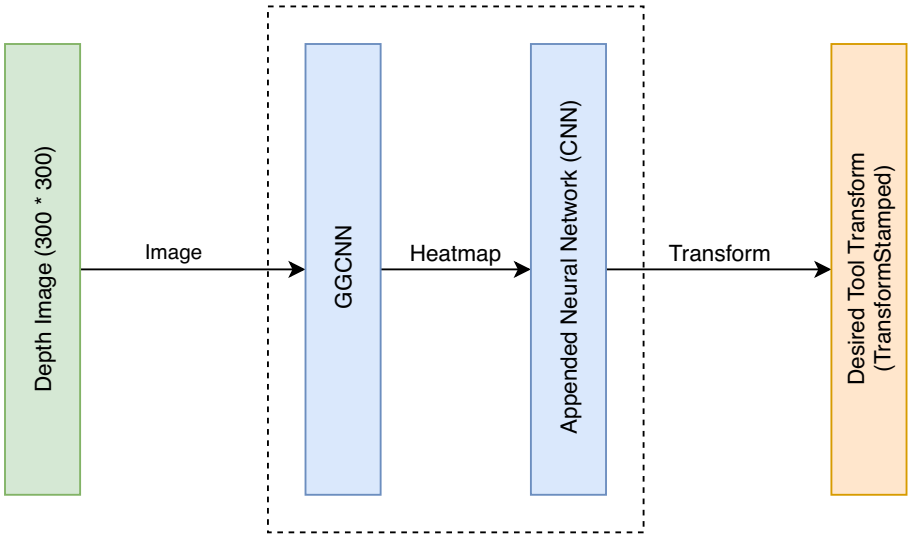

Figure 6.6: Enhanced GGCNN Overview

7 Future Work

Some ideas on future work:

1. Bug fix: if time-out is set very long and no new tool transforms are published to the KUKA control node, the robot stays in the last position which it successfully received. Once new information is published on the topic the robot tries to move there immediately using a real time motion. This may cause problems when the jump is too big (hence we should rather do a PTP movement if the jump is larger than a threshold). For now either leave the default time-out and the robot will move safely to its home position once the signal is lost or be sure to publish a pose which is close to the one where you left off.
2. Evaluate accuracy of AprilTag tracking using RGB optics. Is it sufficient for autonomous grasping?
3. Calculate transform from depth optical frame to tag using approach in section **??** and compare accuracy with RGB optics approach.
4. Control two robots using one Leap Motion. Only thing that needs to be changed is that two interaction boxes need to be formed and both hands need to be processed (not only hand which is visible the longest).
5. Further investigation of autonomous robotic grasping. Test the proposed approach to improve GGCNN using a human in the loop.
6. Integration of force detection in robotic hand node. Limit grasping once pressure threshold reached.
7. Depending on KST advancement, publish KUKA status in real-time mode. The KST currently only supports reliable status information (i.e. "getters") in PTP mode and not during direct servoing.
8. Make more complex tool transforms possible (full (4,4) transformation matrix in configuration file) and calculate home pose accordingly.
9. Determine virtual Leap Motion transform relative to tool home transform and not KUKA base frame. Advantage is that this will adjust the virtual Leap Motion position if a custom home pose is given in the configuration file. This would make the system even more modular (e.g. for KUKA robots that are mounted on the ceiling or the wall).

10. Visualise interaction box, give users feedback if they are holding their hand in a valid starting position and orientation.

11. Use calibrated DH parameters. The DH parameters used by the KST can be changed in the function `constDhDataOf7R800.m` of the KUKA Sunrise Toolbox.