OSNABRÜCK UNIVERSITY

BACHELOR THESIS

# Development of a Linguistic Human-Machine Interface to a Kitchen Robot for the Manipulation of Objects

*Author:*
Julian GAAL

*Registration number:*
966047

*1st Supervisor:*
Dr. Joachim HERTZBERG

*2nd Supervisor:*
Dr. Alessandro SAFFIOTTI

*A thesis submitted in fulfillment of the requirements for the degree of Cognitive Science B.Sc.*

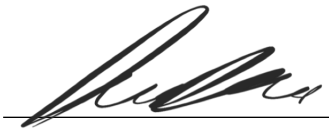
*in the*

ReGround | Knowledge-Based Systems
In cooperation with AASS, Örebro University, Sweden

May 15, 2019

# Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.



signature

Osnabrück, May 15, 2019

city, date

OSNABRÜCK UNIVERSITY

# *Abstract*

Institute of Cognitive Science
In cooperation with AASS, Örebro University, Sweden

Cognitive Science B.Sc.

**Development of a Linguistic Human-Machine Interface to a Kitchen Robot for the Manipulation of Objects**

by Julian GAAL

Symbol grounding refers to the process of associating symbols from language with a corresponding object in its environment. In contrast to previous research in this field, *relational* symbol grounding considers the full environment with multiple objects and their properties, especially relationships among objects, for determining these associations.

This thesis was completed as part of an international research collaboration, ReGround project[1], and integrates the work on relational symbol grounding and the interpretation of linguistic expressions, done by other partners, with object manipulation (grasping, dropping) performed by a robotic arm. The goal is to create a ROS[2] framework that processes an expression to determine the requested object and instruction, and executing a robotic motion accordingly. The system was shown to be reliable overall and allowed for an object grasping success rate of above 95%.

[1] ReGround project: http://reground.cs.kuleuven.be/Abstract
[2] ROS: http://www.ros.org/

# Contents

**Declaration of Authorship** i

**Abstract** ii

**1 Introduction** 1
- 1.1 Context and Motivation . . . 1
- 1.2 Objectives and Implications . . . 2

**2 Workspace and Software Stack** 3
- 2.1 ROS - Robotic Operating System . . . 3
- 2.2 Physical Workspace . . . 5
- 2.3 MoveIt! . . . 6
- 2.4 Language and Relational Symbol Grounding . . . 8
- 2.5 Anchoring and Relational Tracking . . . 9
- 2.6 Definitions . . . 10

**3 Robotic Arm Manipulation** 12
- 3.1 Motivation and Goals . . . 12
- 3.2 Pose Generator . . . 13
  - 3.2.1 Adjusting gripper orientation . . . 14
  - 3.2.2 Adjusting height and position . . . 15
- 3.3 ROS Actionlib Server and Client API . . . 16
  - 3.3.1 Jaco Manipulation Server . . . 16
  - 3.3.2 ROS Interface: Jaco Manipulation Client . . . 18

**4 Integration** 21
- 4.1 ReGround Pipeline . . . 21
- 4.2 Reground Wrapper . . . 25
- 4.3 Adaptations . . . 26
  - 4.3.1 Anchoring System . . . 26
  - 4.3.2 Obstacle Avoidance . . . 28
- 4.4 ROSSpeechPublisher . . . 32

**5 Testing and Results** 33
- 5.1 Background . . . 33
- 5.2 Recording . . . 34
- 5.3 Analysis . . . 34
- 5.4 Result Types . . . 35
- 5.5 Baseline Test . . . 36
- 5.6 Anchoring Test . . . 37
  - 5.6.1 Results Explored . . . 38
- 5.7 Improvements . . . 41
- 5.8 Obstacle/Occlusion Anchoring . . . 44

5.9 Reproducibility . . . . . 46

**6 Conclusions** **47**
6.1 Summary of Achievements . . . . . 47
6.2 Discussion and Future Improvements . . . . . 48
6.2.1 Artificial Constraints and Flexible Planning . . . . . 48
6.2.2 Grasp Pose Generator . . . . . 49
6.2.3 Octomap Obstacle Avoidance . . . . . 49
6.2.4 Hardware . . . . . 49

**Bibliography** **51**

**A Detailed Installation Instructions** **52**

**B Reproducing Tests** **54**
B.1 Anchoring Test . . . . . 54
B.2 Baseline Test . . . . . 55
B.3 Obstacle/Occlusion Test . . . . . 56
B.4 client_grasp_test.cpp . . . . . 57
B.5 client_test.cpp . . . . . 57

**C Video Recordings** **59**

# List of Figures

2.1 ReGround Workspace . . . . . 6
2.2 ReGround ROS TF Schematic . . . . . 7
2.3 Exemplary MoveIt! Planning Scene . . . . . 7

3.1 Robotic Arm Resting Position . . . . . 13
3.2 Gripper Orientation Schematic and Implementation . . . . . 14

4.1 ReGround Software Stack . . . . . 21
4.2 View from Microsoft Kinect . . . . . 23
4.3 Failed Obstacle Recognition . . . . . 24
4.4 Object orientation and size estimation . . . . . 27

5.1 Baseline Test - Poses and Gripping Result . . . . . 37
5.2 Anchoring Test - Poses and Gripping Result . . . . . 38
5.3 Gripper Schematic - Grasping Error . . . . . 39
5.4 Effect of path on gripping . . . . . 40
5.5 Improvements: RRTConnect planning time 1.5 seconds . . . . . 41
5.6 Planner RRT*: 1.5 seconds planning time . . . . . 43
5.7 Planner RRT*: 3 seconds planning time . . . . . 43
5.8 Planner RRT*: 6 seconds planning time . . . . . 43
5.9 Planner RRT*: 9 seconds planning time . . . . . 43
5.10 Original Anchoring Test with Pre-Grasp Pose . . . . . 44
5.11 Exemplary surface occlusion for anchoring test . . . . . 45

# List of Tables

4.1 ReGround Logic Abstraction . . . . . 25

# List of Abbreviations

| | |
|---|---|
| **ROS** | **R**obotic **O**perating **S**ystem |
| **API** | **A**pplication **P**rogramming **I**nterface |
| **URDF** | **U**nified **R**obot **D**escription **F**ormat |
| **RGB-D** | **R**ed-**G**reen-**B**lue-**D**epth |
| **PCL** | **P**oint **C**loud **L**ibrary |
| **OMPL** | **O**pen **M**otion **P**lanning **L**ibrary |

# Chapter 1

# Introduction

## 1.1 Context and Motivation

Imagine being in the possession of a household robot that can be operated by linguistic instructions, a kitchen robot that assists in the preparation of meals, for example. It could assist in whipping cream, grab fresh fruit off shelves or move used kitchen utensils out of the way. Not only would this take care over presumably boring kitchen tasks, but it would also significantly speed up the time of meal preparation. It is essential for such robot to adapt to both the environment and human language for the correct execution of commands. *Thus it needs to identify which objects, actions and relationships in the real world the phrases refer to in order to obtain a true understanding of the world and the complex interactions that govern the physical environment.*[1]

This process is often referred to as symbol grounding, the association of symbols from language with matching objects in the environment, but has rarely included the understanding of spatial relations between objects and its full environment. It has been traditionally used to identify single objects and their inherent properties. The *relational* symbol grounding technology developed as part of the ReGround project proposes that *the grounding process must consider the full context of the environment, including multiple objects, their properties, and relationships among these objects* (Antanas et al., 2017). This novel approach is capable of mapping linguistic geometric relations like "to the front of", "to the left of" or "behind" to recognized object in the environment.

[1] ReGround project website: https://archive.fo/Bz8aU (archived 30.04.2019)

## 1.2 Objectives and Implications

This thesis discusses the prototyping of a robotic arm system that can be interactively operated via linguistic instructions and expressions. These interactions are simple at this stage of the project, and are limited to grasping and dropping detected objects. Most importantly, it focuses on the implementation and extensive testing of a framework, built with ROS[2], that integrates necessary components like object and language recognition, and finally the relational symbol grounding technology itself into a MoveIt![3] configured robotic arm.

This process provides crucial feedback to the current ReGround team members as the use of the technology has been restricted to simulated and highly controlled environments before this thesis.

Lastly, extensive testing has been performed to allow for an accurate assessment of the maturity of the framework and each component that makes up the relational symbol grounding technology.

[2]ROS: https://archive.fo/cPdvw (archived 30.04.2019)
[3]MoveIt!:
https://web.archive.org/web/20190330210122/https://moveit.ros.org/ (archived 30.03.2019)

# Chapter 2

# Workspace and Software Stack

The following introduces the background technologies necessary to understand the following chapters and are not part of the contributions of this theses.

## 2.1 ROS - Robotic Operating System

This project uses ROS[1] for the interaction between all necessary components, ranging from the vision system to the robotic arm manipulation itself. To explain its core idea, it is best to quote the official documentation: *"ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers."*[2] While the ROS ecosystem is extensive, I would like to explain a few concepts necessary to understand this paper.

**Nodes** are units that perform computation. A ROS code base may contain many Nodes that each perform different calculations in a distributed fashion.

ROS implements a **Publish-Subscribe model**. Publishers are identified by its name, the *topic*, typically prepended by the / character, e.g. */a_topic*, and *publishes* data. A Subscriber can receive, or *subscribe to* all data, or *messages*, published by a specific topic. ROS offers a common message interface[3], but allows the use of custom messages, with minor limitations.

[1] ROS: https://archive.fo/cPdvw (archived 30.4.2019)
[2] ROS description: https://archive.fo/6yMnG (archived 09.11.2013)
[3] ROS messages:https://archive.fo/6YGQf (archived 30.04.2019)

A **Package** includes Nodes, configuration files, documentation and more, and represent the most granular thing one can build in the ROS ecosystem.

**Frames** are essential when working with *URDF*, to describe the relations between different robot elements in space. Each frame represents a *local* coordinate system in a single *global* coordinate system, often called *base_link* in ROS. Every sensor has its own local coordinate system, or frame, but needs to be *connected*, so to speak, to other parts of the robot by providing the translation and rotation to the base frame. For example, a lidar scanner in frame *laser* may be mounted with translation $\tau$ and rotation $\rho$ in relation to frame *base_link*. This information allows the ROS package *tf* (explained shortly) to provide the relations and distances between all elements of the robot in real time, even when parts of the robot are moving. Frames are noted in snake case, and will be used in `this_notation` across this document.

A **Catkin workspace** is a ROS workspace that may contain all elements mentioned above, and more. *Catkin* stands for the integration of ROS' build tool by the same name. It is the central organizational structure, is used to invoke the Catkin build system, create new packages, and launch ROS nodes and packages.

**RViz** is a visualization tool for ROS. It enables the visualization of point clouds, geometric shapes (e.g. for obstacles), a model of the robot itself, maps and more. It is frequently used to oversee the behavior of Nodes that are hard to interpret with raw data alone and is essential for debugging and MoveIt! (see section 2.3).

**Actionlib**[4] is an interface to create and interact with preemptable tasks in ROS. It creates a local *Action Server* that executes a sequence of commands, in the order they where received in from an *Action Client*, but is accessible during the entire time of execution. This enables a user *"to cancel the request during execution or get periodic feedback about how the request is progressing"*.[5]

**Services** provide features similar to Actionlib, with one big difference however: once a request is sent, there is no way to interrupt execution or receives status information. Services are used for very small preemptable tasks.

[4] ROS Actionlib: https://archive.fo/veRHs (archived 30.05.2019)
[5] See footnote 4

**tf** is essential to every ROS workspace and keeps track of all translations and rotations during the time of execution and is used to periodical request the transformations between frames to calculate poses, trajectories and more. These relations are stored in a *tf tree*.

**Launch files**[6] are files that can be used to start multiple ROS nodes from one file. They allow setting specific parameters for each node and are central organizational units of a ROS workspace.

These elements can be used to exploit the single most useful feature of ROS: modularity. Nodes and packages can be launched from different computers, each implemented in a different programming language and work together seamlessly. Of course, ROS cannot make real-time guarantees for certain robotic fields like aerial robotics, but it has sped up robotic development dramatically, especially in academia. For more information, visit `www.ros.org`.

## 2.2 Physical Workspace

The workspace is aimed at resembling a kitchen tabletop. Mounted to it is an RGB-D depth vision system, the Microsoft Kinect V2[7], and a robotic arm with 7 degrees of freedom and a three fingered gripper, Jaco v2 by Kinova[8]. While the robotic arm is mounted to the left of the future cook, the depth sensor is mounted on the right edge at a height of about 60cm above the table top. The active workspace is around 65x65cm in size. The workspace computer is running an Intel i7 processor and an NVidia GTX 970 graphics card to be able to handle the multitude of ROS packages adequately. Frames, marked in red, will be used at a later point in time to explain the implementations of components in detail. While the root of the ROS system is represented by the frame *base_link*, all other frames are connected to it, either directly, or indirectly. This is necessary for coordinate transformations and motion planning. Please note that ROS follows the right hand rule[9] for its coordinate frames.

[6] ROS launch files: https://archive.fo/ATnCU (archived 30.04.2019)
[7] Microsoft Kinect V2: https://archive.fo/XetJs (archived 30.04.2019)
[8] Jaco v2 product overview: https://github.com/juliangaal/jaco_manipulation/blob/jaco2_reground/jacov2.pdf (visited 30.04.2019)
[9] ROS Standards Guidelines: https://archive.fo/JRljw, section "Chirality" (archived 03.05.2019)

**Figure 2.1:** ReGround workspace

## 2.3 MoveIt!

MoveIt! is a popular motion planning framework. In robotics, motion planning refers to translating a movement task into explicit motions that satisfy spacial constraints, for example constraints by obstacles in the environment. MoveIt! is used extensively in robotics research, especially involving robotic arms. It incorporates *"the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation"*[10] and was used to configure and manipulate Jaco2. In a nutshell, MoveIt! uses the robots physical characteristics like location of the motor joints and dimension information defined in the URDF format to provide motion planning, navigation and control while preventing self-collisions. An existing configuration for Jaco2, provided by the manufacturer, was adapted and improved for the use in this project. Furthermore, the available RViz plug-in allows the visualization of collision objects, the robot and planned trajectories for visual control via the **Planning Scene**, which will be mentioned frequently as well. It *"represents all the information needed to compute motion plans: The robot's current state, its representation (geometric,*

[10]MoveIt!: https://web.archive.org/web/20190330210122/https://moveit.ros.org/ (archived 30.04.2019)

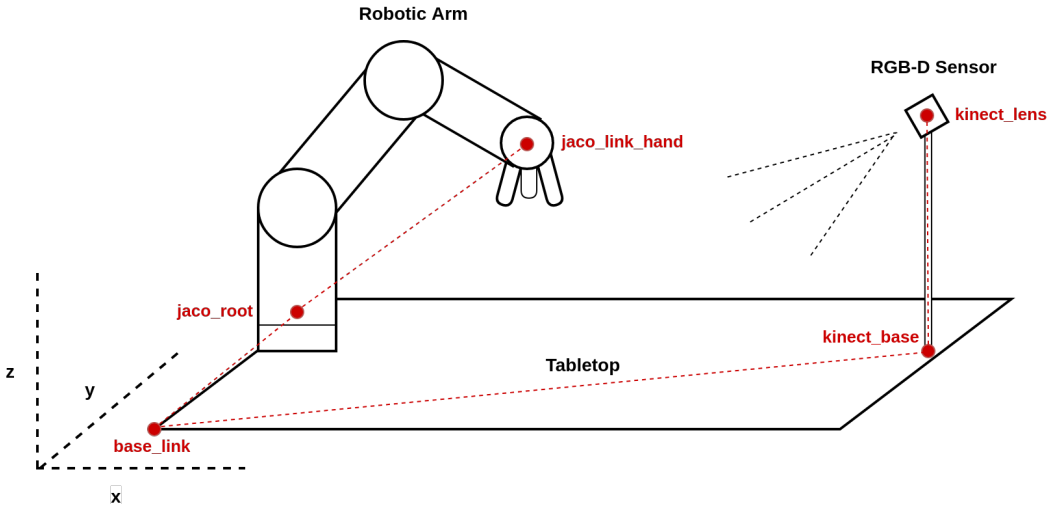



**Figure 2.2:** ReGround workspace: sensor and tf frames schematic

*kinematic, dynamic) and the world representation. Using this information, things like forward kinematics, inverse kinematics, evaluation of constraints, collision checking, are all possible."* [11]. However, for the end user (and reader) it represents the visual control interface in RViz first and foremost and will be mentioned as such. An example for this interface can be seen in figure 2.3.

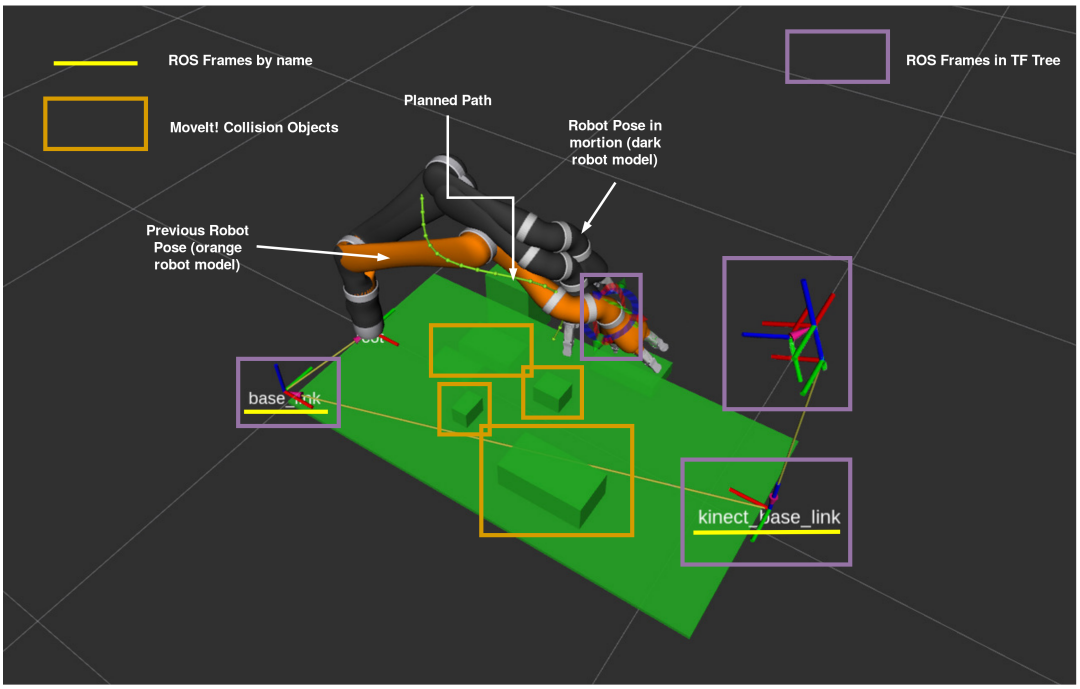



**Figure 2.3:** Exemplary MoveIt! Planning Scene with frames, obstacles, planned trajectory and the robot model itself

[11] Planning Scene description: https://archive.fo/jGiax (archived 30.04.2019)

MoveIt! is available as a ROS package and integrates seamlessly into the Catkin build system and workspace.

## 2.4 Language and Relational Symbol Grounding

A critical component for the interaction of relational symbol grounding components and the robot system is language processing, as it is the very first and only interface to the kitchen robot a user has access to. The following requirements are to be met

**Human spatial understanding**: In a kitchen setting, or human setting for that matter, spatial relations are critical for orientation and object manipulation. Therefore the language system needs to be able to generate and understand geometric positions from commands containing spatial relations, e.g. "Pick up the egg to the right of knife".

**Natural language input:** The language interface needs to seamlessly blend into human conversation to prevent a "robotic feel", e.g.

*"Pick up the egg. Drop it into the pot"* or

*"Pick up the green pepper and drop it onto the cutting board"*

ReGround *"developed a system to ground the meaning of words onto physical objects, their properties and relations between them. The meaning of nouns, adjectives and prepositions are learned as neural components."* Most importantly however, having learned *"the meaning of names, spatial prepositions, and navigational action verbs"*[12] allows for the execution of commands similar to the exemplary commands just mentioned, in accordance with Arkan Can et al., 2019.

It is available as a ROS package which provides a ROS Service for interaction: after receiving a text command in line with the examples above, the Service returns the anchor id of the best matching anchor for further processing from an Anchoring System, described below. In the case of an drop instruction, a drop position is generated with the given geometric relation and position estimate in mind. It can be downloaded at `https://bitbucket.org/reground/language/src`.

[12] ReGround specification: https://archive.fo/Bz8aU (archived 30.04.2019)

The language system will be presented in great detail at SpLU-RoboNLP 2019[13].

## 2.5 Anchoring and Relational Tracking

Anchoring *"is the problem of connecting, inside an artificial system, symbols and sensor data that refer to the same physical objects in the external world"* (Coradeschi and Saffiotti, 2003). These *symbols* are best thought of as labels that denote objects, often stemming from image classification systems. *Anchors* refer to objects and their respective label, whose correspondence to physical objects has been established and is maintained. This necessarily includes an image or object recognition system, as well as a symbolic reasoning component.

By matching anchors like color, size or shape to existing anchors, anchoring is capable of reliably tracking objects even in reasonably dynamic environment. The following requirements are to be met

**Anchoring Accuracy:** a cook talking to his kitchen helper has to be confident in the anchoring in order to be confident in giving commands. The descriptor of an object can not be allowed to change during the receiving process of a command: Imagine a scenario in which you want the robot to pick up an egg, but a split second before entering the command the anchoring system begins to recognize the egg as a ball. Missing or incorrect anchoring information will prevent the robot from executing the command - if at all.

**Size and Location Estimation Accuracy:** successful grasping and dropping depends heavily on an accurate size estimate of recognized objects as this is tied directly to the generation of a bounding box around said object. This box is crucial for integration with the MoveIt! framework. Inconsistencies in size can lead to incorrect detection of collisions and failed path planning attempts that are hard to debug, even in cases, where the location estimate was correct.
Additionally, obstacle avoidance as implemented in section 5.8 relies on accurate size estimation for successful collision avoidance between the robotic arm and objects on the simulated kitchen top.

[13]SpLU-RoboNLP 2019: https://web.archive.org/web/20190430085238/https://splu-robonlp.github.io/ (archived 30.04.2019)

**Relational Tracking:** Having done location estimation, it is necessary to track *relations* between objects over time, as well, as a form of *relation tracking*. This forms the base of necessary knowledge to allow for *relational symbol grounding*, briefly described in section 2.4.

This project uses an anchoring and object recognition system *"which handles the full processing pipeline of segmenting objects, extracting features, classifying objects, ground perceptual-symbolic correspondences, and anchoring objects."*[14]. The systems semantically maps the environment, in which *"anchored objects are retained by preserving relations through probabilistic reasoning"* (Persson et al., 2019). It integrates into the ROS workspace as a ROS package, and provides an Action Server for information requests about the current anchors, their dimensions, relations to each other, labels and more. It is available at `https://bitbucket.org/reground/anchoring/src`.

Object classification is realized by a deep convolutional neural network, *"which has been trained and fine-tuned based on categories of objects that can be expected to be found in a kitchen."*[15] Each recognized object is assigned a label and an anchoring id. While the latter is used to track the object over time, the former is determined by the image classification system. The label is subject to significantly more frequent changes than the anchoring id itself, as the anchoring id only changes when the system loses track of the object, while the label changes every time the classification system recognizes an object as another, due to changes in visual features. Additionally, its size estimate is represented by a rectangular bounding box around the object.
Lastly, the centroid of the front face of an object is provided as the location estimate.

## 2.6 Definitions

A **point cloud** is a set of data points in space that represent objects in three dimensions, provided they are in sight. They are generally produced by 3D depth sensors, for example infrared based sensors like the Microsoft Kinect [16], which is in use in this project.

[14] ReGround specification: https://archive.fo/Bz8aU (archived 30.04.2019)
[15] see footnote 14
[16] Microsoft Kinect: https://archive.fo/XetJs (archived 30.04.2019)

A **RGB-D Sensor**, or Red-Green-Blue-Depth Sensor, is a (video) camera that is able to augment the conventional RGB image with depth information, typically from an infrared based depth sensor. They are popular because of their small footprint and low cost.

A **Pose** [17] is a representation of a point, with x, y and z coordinates, and it's rotation, a quaternion.

The use of **Language System** will refer to the Language Processing interface described in section 2.4

The use of **Anchoring System** will refer to the Relational Anchoring ROS package described in section 2.5.

[17] Pose: https://archive.fo/xt5rI (archived 30.04.2019)

# Chapter 3

# Robotic Arm Manipulation

This chapter discusses the majority of my direct contribution to the ReGround project: Building an extensible, general purpose interface for the manipulation of objects with the robotic arm. A well designed interface is crucial for the ReGround project and allows for more confident and purposeful decisions in the integration process of existing ReGround components, described in chapter 4. The code is available as the ROS package Jaco Manipulation[1]

## 3.1 Motivation and Goals

While MoveIt! provides a very helpful low level API to Jaco, there are many configuration options that leave the door open for user errors. In practice, this resulted in numerous errors during motion planning and execution from MoveIt!. Furthermore, in the process of testing this interface, decimal-point mistakes in the position the robot was commanded to move to were sometimes hard to trace back to being a user error, as these mistakes resulted in only generic planning errors in MoveIt!.

The goal is to build an higher-level API for the interaction with the robotic arm that minimizes possible user errors by hiding implementation details, while offering an elegant interface that can be used in the ReGround pipeline by the Anchoring and Language Grounding System. Therefore, it is necessary to implement a system that generates possible poses by taking the centroid and size estimation of the target object by the Anchoring System into account. Additionally, the degree of which the gripper closes must be adjusted to reflect the objects size.

[1]Jaco Manipulation: https://github.com/juliangaal/jaco_manipulation

## 3.2 Pose Generator

A central aspect to Jaco's usefulness as a kitchen robot is its ability to grasp objects. To ensure easy extension and modularity in the future development of the ReGround project the generation of target poses is computed in a separate library that adjusts various parameters of the incoming pose to accommodate for objects seen by the vision system. This library receives a pose in `base_link` and a bounding box describing the target object, while generating a target pose in the `jaco_root` frame, which refers to the planning frame of the MoveIt! framework. Poses in the planning frame refer to the pose of `jaco_link_hand` in space, which represents the "wrist" of the gripper.

Though work on grasping objects from a frontal position is not completed, it provides functionality for grasping and dropping from a top position.

**Figure 3.1:** Jacos resting position

### 3.2.1 Adjusting gripper orientation

Any adjustment in gripper orientation needs to be achieved by manipulating the local Cartesian coordinate system of frame *jaco_link_hand*. Consequently, it was deemed easier to work with orthogonal unit vectors describing the required local Cartesian coordinate system. Only later was it converted to match the Pose specification.

While generating an orientation for the target pose, there is only one crucial restriction: the z-axis of this has to be exactly vertical to ensure a top grasp. The unit vector describing the the z-axis is trivial.

$$\vec{z} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$$

Looking at Jacos resting position (figure 3.1), the initial idea revolved around adjusting the orientation in a way in which the thumb gripper always faces the root of the robotic arm. Upon preliminary testing, this seemed like a good starting point with a high probability of optimal planned trajectories. The direction of the vector

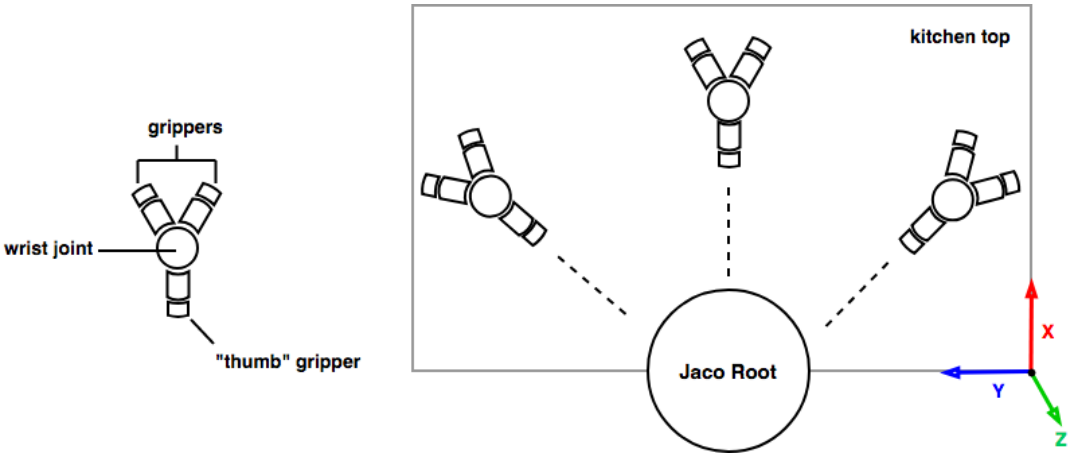



**Figure 3.2:** Orientation implementation effect, birds eye view

pointing from Jacos root to the thumb, $\vec{x}$ can be acquired by normalizing the vector containing both x and y coordinates, excluding z, as we are only interested in an exactly horizontal direction. Calculating the cross product of $\vec{x}$ and $\vec{z}$ results in the remaining y-axis.

$$\vec{y} = \vec{z} \times \vec{x}$$

$$\vec{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} \frac{x}{\sqrt{(x)^2+(y)^2}} \\ \frac{y}{\sqrt{(y)^2+(x)^2}} \\ 0 \end{bmatrix}$$

These three vector span a 3x3 matrix, that can be converted to a quaternion with ROS tf, in order to match the Pose specification.

In summary, this development process and experimentation helped immensely for later ease of modifications, which will be discussed in the integration section in 4.3.1.

### 3.2.2 Adjusting height and position

While, due to mounting constraints, Jacos `jaco_root` is mounted 5 cm higher than `base_link`. `jaco_link_hand` represents the last frame in the robotic arm, seen from bottom to top, and "wrist" of the gripper. The height of a *grasp pose* is adjusted in multiple stages.

The *maximally low height* is calculated to guarantee a pose free from collisions with the work surface, or table. This can be achieved by making use of transformations between `base_link` and `jaco_root`, held by ROS tf. In `base_link`, the bare minimum height of any pose was measured, by hand, to be 20 cm (the distance between `base_link` and `jaco_link_hand`), with the gripper barely touching the table. Using this information, the initial height in the planning frame `jaco_root` is set by subtracting the height translation between these two frames from the minimum height and adding 2.5 cm to ensure no contact between the grippers and the table. This way provides a more flexible way of height generation: Because it uses the height difference information from ROS tf, remounting the robotic arm will not break the pose generation, granted the ROS frames are updated to reflect the mounting changes. This 2.5cm offset is necessary because the individual fingers reach a slightly lower position, relative to its resting position, while closing or opening due to changes in the finger joints.

*Adjustments for the bounding box* representing the target object are calculated next. The position of the target object is described by its centroid, meaning the actual

highest point in the target object is $height * 0.5$ higher than the centroid.

Additionally, *a static offset* is added to accommodate for the distance between the link *jaco_link_hand*, repre and the grippers finger tips.

*Lastly*, the height must be adjusted according to the size of the target object. Due to physical limits, the gripper on Jaco can lift objects with a maximum width and length of 13 cm. While the lowest (initial) height can be reached when the target object has a width and height of 6 cm, the additional height adjustment needed for larger objects can be described by the linear function $f(x) = 0.94x + 13.23$, where x amounts to the greater value of length or width in centimeters.

Generating a pose where an object will be dropped, as *drop pose*, is generated in the same way as described in section 3.2.1, while the height adjustment proceeds with the first two adjustments just discussed. After this, a static offset is added to achieve a dropping height.

## 3.3 ROS Actionlib Server and Client API

### 3.3.1 Jaco Manipulation Server

Jaco Manipulation Server is a ROS Actionlib Server that hides all implementation details of the calls to the MoveIt! API and receives all commands from Jaco Manipulation Client (section 3.3.2). This includes the planning and execution of received target positions, as well as the updating of the visuals in the Planning Scene in RViz. Multiple parameters are exposed to allow for quick modification of tests and easier debugging. The following can be adjusted as part of the launch file[2]:

- `planner_id`: Set the type of trajectory planner[3] MoveIt! will use to plan. The id must be chosen from the configuration file[4]. More on the planners used in this project can be found in section 5.6.1.

[2]Jaco Manipulation Launch File: https://github.com/juliangaal/jaco_manipulation/blob/jaco2_reground/launch/jaco_manipulation.launch (visited 01.04.2019)
[3]Available MoveIt! planners: https://web.archive.org/web/20190330204532/https://moveit.ros.org/documentation/planners/ (archived 30.03.2019)
[4]Jaco Manipulation planner: https://github.com/juliangaal/wpi_jaco/blob/develop/jaco_moveit_config/config/ompl_planning.yaml (visited 01.04.2019)

- `planning_time`: The maximum amount of time MoveIt! is allowed to plan before aborting.
- `planning_attempts`: The amount of simultaneous planning attempts. Due to randomness in planners it is advised to use multiple. MoveIt! will chose the best available plan.
- `allow_replanning`: Specify whether the robot is allowed to re-plan in case of detected changes in the environment. This can be useful when using integrated depth sensors to create an Octomap, as discussed in section 4.3.2.
- `pub_debug`: Publishes information necessary for testing described in section 5.2 on topic `/jaco_manipulation/debug/`.

Additionally, the amount the gripper moves to enclose the target object is adjusted according to the size of the bounding box received. The gripper closes at a value of 0.0, while being opened to its maximum capacity at a value of 6500.0. A slope of -433.0 approximates the amount of closing motion needed for an object 1 cm smaller than the maximum, an object with an diameter of 13 cm. These factors result in this linear function describing the relationship between object size and gripper closing motion, where x stands for the diameter of the target object in centimeters.

$$f(x) = -433.0x + 6500.0$$

ROS Action Server is designed to be a permanent communication endpoint that can be called from multiple ROS instances and nodes by the client. It can accept custom goal types, described next.

### 3.3.2 ROS Interface: Jaco Manipulation Client

The ROS Actionlib Client, Jaco Manipulation Client, is provided to the user as an interface to Jaco Manipulation Server and most importantly for ease of use and safety, by hiding many implementation details of movement execution goals: in order to avoid arm trajectory planning errors by carelessness alone, the user has the ability to set multiple types of goals that are sent to Jaco Manipulation Server after processing. These make sure to only expose goal parameters that have to be changed in the live environment. In total, there are 5 goal types to choose from

- *Pose Goal*: a goal that moves `jaco_link_hand` to the requested Pose.
- *MoveIt! Config Goal*: MoveIt! allows the user to set predefined, fixed joint states in its configuration files. Typically, these are used to define a "Home" position for the robotic arm in question, or to test multiple joint states. The "Home" MoveIt! Config Goal can be found in the configuration file[5], executed in figure 3.1
- *Joint State Goal*: Each joint of Jaco2 an be set individually. This goal sets the rotation of each joint to the determined position
- *Grasp Pose Goal*: A *grasp pose* followed by a grasping attempt. This goal is designed *specifically* with ReGround in mind: the pose is adjusted by the Pose Generator to grasp a target object defined by its bounding box, which is provided by the Anchoring System, and performs a small lift, in accordance with the ReGround specifications shown in section 4.2.
- *Drop Pose Goal*: A *drop pose*, generated by the Pose Generator, followed by the dropping of the object in hand. Lastly, the arm moves to its resting position (fig 3.2).

Each goal inherits crucial properties required by MoveIt! from a base goal class called *MoveItGoal*. This way it is made sure that every goal sent to the ROS Actionlib Server is valid and handled accordingly. This example is meant to provide you with an idea of the level of abstraction provided to the user.

[5]MoveIt! config file for Jaco: https://github.com/juliangaal/wpi_jaco/blob/develop/jaco_moveit_config/config/jaco.srdf#L32 (visited 01.10.2019)

```
#include <jaco_manipulation/client/jaco_manipulation_client.h>

using namespace jaco_manipulation;

int main(int argn, char *args[]) {

    ros::init(argn, args, "pam_client");

    client::JacoManipulationClient jmc;

    BoundingBox b;
    b.header.frame_id = "base_link";
    b.description = "bottle"; // description optional
    b.point.x = 0.695;
    b.point.y = 0.3;
    b.point.z = 0.105;
    b.dimensions.x = 0.06;
    b.dimensions.y = 0.06;
    b.dimensions.z = 0.21;
    jmc.graspAt(b);

    jmc.moveTo("home");
}
```

It shows two goal types: a "bottle" defined by its `BoundingBox` that will be grasped by the gripper as part of a *Grasp Pose Goal*, and a command to move `home`, which is predefined in the MoveIt! config and executed as a *MoveIt! Config Goal*. The client will handle the conversion from custom goal to a goal compatible with the MoveIt! framework by generating a grasp pose (section 3.2 and setting all necessary definitions before calling the server, where execution is handled.
Furthermore, because a tiny accidental slip in the decimal place of a location or object size value can mean that the target pose physically can't be reached and can go unnoticed, an additional layer of security was added to the API: the chance of encountering decimal-point mistakes was reduced significantly by making use of a C++11 feature called *user-defined literals*[6]. It allows for the definition of custom suffixes and can be used, for example, to ensure the safe usage of different units. In the Jaco Manipulation Client it is used for the units *meters*, *centimeters* and *millimeters*. An example:

```
float f = 0.1f          // built-in literal suffix

BoundingBox b;
 ...
b.point.x = 0.6_m    // user-defined literal _m: compiles to 0.6
b.point.y = 60.0_cm  // user-defined literal _cm: compiles to 0.6
b.point.z = 600.0_mm // user-defined literal _mm: compiles to 0.6
 ...
```

[6] User defined literals: https://archive.is/2sRaj (archived 30.04.2019)

User-defined literals have to be prepended by an underscore to ensure that built-in literals are not overwritten. The implication details of these user-defined literals can be found on Github.[7]

It is important to note that these goals will only be *manually* set when testing the robot manipulation itself. In all other cases, objects detected by a recognition system must be translated to fit the `BoundingBox` specification, e.g. seen in chapter 4. Nevertheless, it is crucial to minimize user errors during this process to speed up development of this critical component to the ReGround project and ensure reliability. In the fully functional ReGround pipeline, the bounding box will be provided by the anchoring system.

More examples are provided[8].

[7] User-defined literals implementation: (visited 30.04.2019)
github.com/juliangaal/jaco_manipulation/blob/jaco2_reground/include/jaco_manipulation/units.h

[8] API examples: https://github.com/juliangaal/jaco_manipulation#usage-2 (visited 30.04.2019)

# Chapter 4

# Integration

While the previous chapter presented the basic building blocks necessary to integrate and interface to existing code from ReGround team members, this chapter discusses the integration of existing ReGround software and adaptations to my contributions to create a streamlined software pipeline.

## 4.1 ReGround Pipeline

To understand the integration steps, it is helpful to discuss the overall structure of the resulting system first. It describes the system presented in Arkan Can et al., 2019 in an abstract fashion.

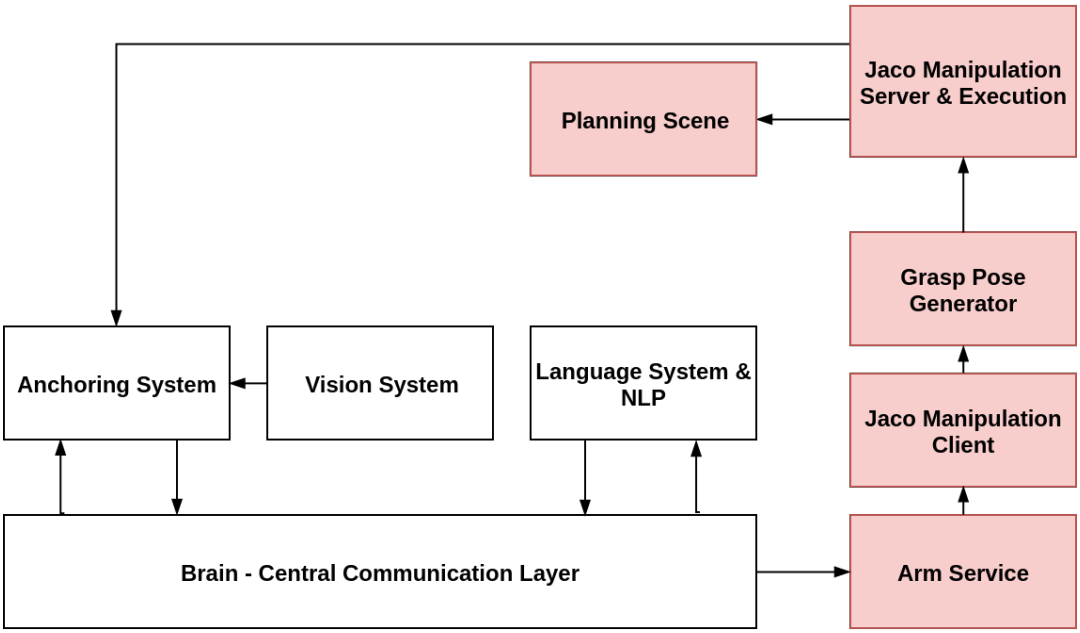



**Figure 4.1:** ReGround software stack structure in its entirety

This visual representation is provided to give thorough overview of the individual components. My main contributions to the project are marked in red, while I adapted parts of the "Brain" to interface with existing software. I would, however, like to go into greater detail to describe parts of the structure. One of the strengths of ROS as an operating system tailored to robotics is modularity, which is beneficial for research projects like this that consists of multiple or existing ROS components, that were developed apart from another for large amounts of time. The "Brain" of the ReGround project binds all parts of the project together. It is a ROS node written in Python that communicates with different ROS Action Services and Actionlib Servers responsible for anchoring, language grounding and arm manipulation. A typical action processed in the brain entails multiple steps.

**Step 1** A command line (or web) interface records the requested command and publishes it for further processing and debugging. For ease of explanation, let's call the topic published to */command*.

**Step 2** The Anchoring System subscribes to */command* and publishes all information about current anchors, including size, shape, color and position, in a format tailored for the Language Grounding System. It is important to publish this information separately, to ensure that the following steps work with the most accurate data. For ease of explanation, this topic shall be called */anchor_info*.

**Step 3** Language grounding subscribes to */command* and */anchor_info* and waits for both incoming messages. For each received message, the system then extracts the target object and geometrical relations from the received text. Finally, it returns the anchor id of the item that best matches the position and name described in the received command.

**Step 4** Next, all robot manipulation elements are executed. The ReGround Wrapper accepts the type of command and anchor id, corrects its pose to represent the absolute centroid, generates a goal with Jaco Manipulation Client and hands this over to the Jaco Manipulation Server for an attempted execution.

**Step 5** Lastly, the anchoring system updates its position estimate for the object referring to the anchor id extracted from the Language Grounding System. This

needs to be done manually because the Anchoring System may lose track of the object while in the gripper of Jaco (see figure 4.3). The orientation of the gripper, generated by the Pose Generator, often results in the objects sight being blocked by 2 fingers grippers at once. While the anchoring system periodically recognizes the object inside of the gripper, it is very hard to separate this object from the grippers. Only after the gripper lets go of the target object and is in clear sight again.

To demonstrate the abilities and interaction with the robot from a more practical and less technical perspective, I would like to guide you through a visual tour of a typical interaction.

A command is entered. Separate, or commands connects with "and" are supported.

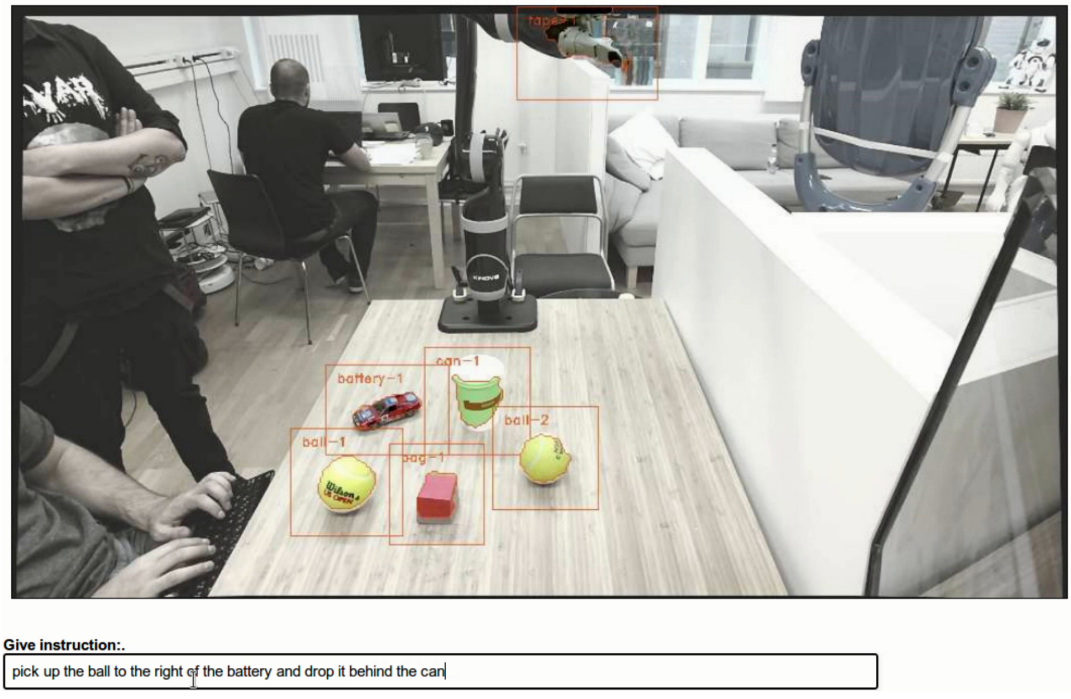
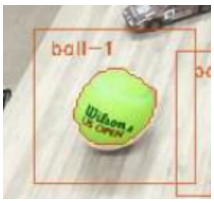



**Figure 4.2:** View from Microsoft Kinect onto work space.

The object to be grasped will be marked in green to allow for visual control.



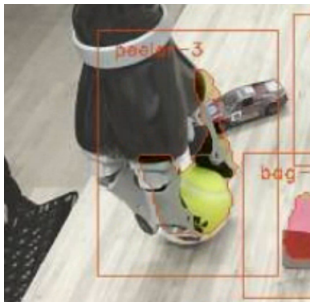
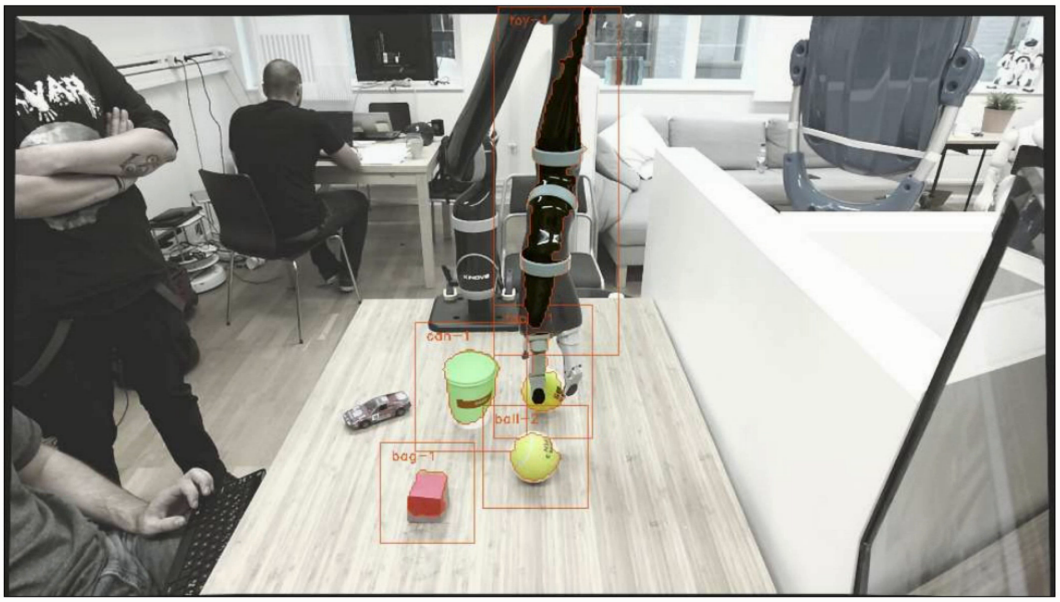


**Figure 4.3:** Failed OR

Next, the object will be grasped. This step significantly reduces the object recognition accuracy. Continuing, the object described in the target command will be visually marked as well.


Finally, the object is dropped in accordance with the command given while avoiding obstacles on its path. Please note that the geometrical relations are processed from the users perspective, *not* the cameras perspective. In this case, the drop is dropped next to the can in the picture above, while it refers to behind from the users perspective. Video recordings of similar interactions with the robotic arm can be found in appendix C.

## 4.2 Reground Wrapper

Early ReGround participants defined a logic layer for all actions in the project, shown in table 4.1.

| **Command** | **Arguments** | **Description** |
|---|---|---|
| PickUp | AnchorID a | Grasp and lift up the physical object denoted by a |
| MoveTo | Position p = (x, y, z) | Move the currently held object at target position p |
| PutDown | Position p = (x, y, z) | Move and release the currently held object at target position p |
| PickAndPlace | AnchorID, Position p = (x, y, z) | Push object a to target position p |
| ... | ... | ... |

**Table 4.1:** ReGround Logic Abstraction

A ROS Service *ArmService*[1] has been implemented to reflect the *Command* column in above table. It is an easily extensible wrapper around Jaco Manipulation Client and is trivial to call from existing code in C++ as well as Python, which was the language of choice for the Brain (figure 4.1). Additionally, the Service acts a translation layer: it converts information received from the Anchoring and Language System to Jaco Manpipulation Client API calls. With increasing complexity, it is important to offer an API for participants of ReGround that adheres to this specification, for multiple reasons:

(a) While Jaco Manipulation Client is a well designed API, it may be hard for team members with less knowledge about the behavior of the robotic arm to use the library directly. It is necessary for testing (chapter 5) to have this level of granular control, but not much else in the current state of ReGround.

(b) Furthermore, it is a standard, i.e. non-ROS, C++ library which would have to be integrated directly into the Anchoring and/or Language System. *ArmService* is a modular ROS Service and standalone ROS node that is accessed from

[1] ArmService Node: https://github.com/juliangaal/arm_service (visited 24.04.2019)

the Brain. The possibility of encountering new and unknown bugs decreases because existing code in the Anchoring and/or Language System stays untouched.

## 4.3 Adaptations

The software stack described in chapter 3 was developed before the official ReGround meeting in Örebro, Sweden. While I developed it to the best of my abilities, the full details and requirements of the system became clear during and after this meeting. This is were this sub-chapter begins: It describes the modifications done to my work and that of team members systems to complete the integration project.

### 4.3.1 Anchoring System

Luckily, the integration of my system into the software stack required very little modification of the existing Anchoring system. Apart from minor position information formatting, the existing code base was used without change as all communication with it is handled via the Brain and ROS Publishers.

Unfortunately however, it does not provide the actual orientation of a target object due to constraints and inaccuracies of the shape estimation. That is, without significant changes: a possible PCL[2] implementation was deemed too extensive and daring at the time. In reality, the Anchoring System estimates two orientations: one perpendicular to the y-axis of the workspace, and one parallel to it, depending on the rotation of the target object. This can be seen in an exemplary manner in figure 4.4 below, though the drawing is exaggerated for clarity. While this issue also results in size estimation errors, it is not clear how far this effect has to be taken to result in gripping failures. Most objects used for testing were circular, e.g. balls, oranges, or rectangular, e.g. carton boxes or a milk carton, and gripping failures were mostly due to height discrepancies in the pose. Therefore, more tests would be necessary to record data for his particular issue and allow for code adaptations. Adapting the Pose Generator to this orientation issue meant dismissing the previous idea of orienting the thumb in accordance with its position relative to Jacos root: the z-axis

[2]PCL: https://archive.fo/xrYqL (archived 30.04.2019)

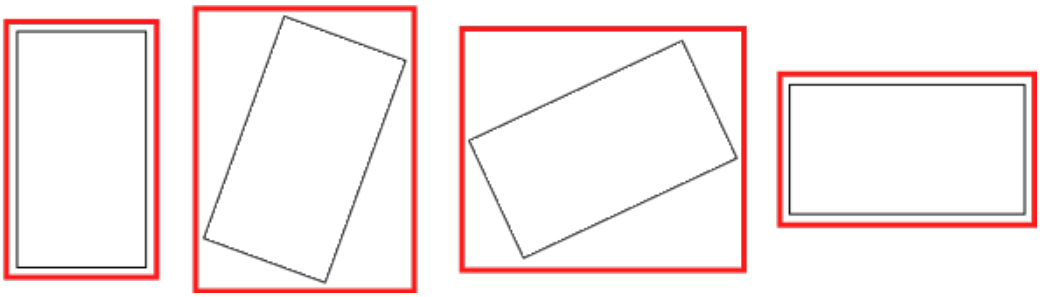


**Figure 4.4:** Object orientation examples and their effect on size estimation and possible overestimation (exaggerated). Size/ Bounding box estimation in red

orientation remains unchanged, while the x and y-axis values in the direction vector are either 1 or 0, depending on the dimensions of the object. Is the length of the target object greater than its width, the thumb must be parallel to the workspaces y-axis. The direction vector can then be described as the unit vector of axis y

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$$

In case the length of the target is smaller than or equal its width, the thumb gripper (see figure 3.2) must be parallel to the x-axis to allow for a successful grasp attempt. Therefore, the direction vector can be described as the unit vector of axis y.

$$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$$

As in the previous iteration, the cross product of these vectors yields the unit vector describing either x- or y-axis, depending on the current rotation. To preserve the top grasp orientation described in the original Pose Generator in section 3.2, the direction vector for the z-axis is simply

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$$

Perhaps most importantly, the Arm Service adjusts the target pose generated by the anchoring system to reflect the actual centroid of the target object, as the anchoring system generates the centroid in relation to the front face of target object, first described in section 2.5. This is done by simply reducing the x coordinate value by half of the estimated width.

### 4.3.2 Obstacle Avoidance

After completing most of the integration work necessary to create the ReGround Pipeline, the need for robust obstacle avoidance became clear: Firstly, to stop the robotic arm knocking over or pushing away the target object on is target trajectory, and secondly to prevent the arm from colliding with other objects while moving the target object in the gripper. In principle, I evaluated two different approaches for obstacle avoidance.

First, the manual approach: The MoveIt! Planning Scene and its API allows direct access to the obstacles that MoveIt! is avoiding by applying constraints to the chosen planner. It is simple and safe to use and is based on the idea of publishing a point at the origin of a defined shape with a standard ROS Publisher. MoveIt! differentiates between

(a) Collision Object: objects that will not or can not be gripped but can be collided with

(b) *Attached* Collision Object: attaches *"the object to the robot so that any component dealing with the robot model knows to account for the attached object, e.g. for collision checking"*[3]

The Planning Scene was utilized by adding the target object as an Attached Collision Object and all other recognized obstacles as Collision Objects to the MoveIt! Planning Scene. In earlier stages of the project only objects that were about to be gripped were added to the planning scene to stop the robotic arm from knocking over this object specifically. However, after encountering significant issues with obstacle avoidance by using the depth sensor (discussed shortly), extending the ROS Action Server to support sending scene objects other than the target scene object from the anchoring system was not a major issue as it allows for re-use of large parts of this existing code. This resulted in the Planning Scene Objects mirroring the exact placements of objects on the simulated kitchen top, while it is updated at the moment of the received grasping command.

[3] Attached Collision Object: https://archive.fo/fRc2B (archived 30.04.2019)

**Advantages** This method requires very little computational resources and allows for *immediate* visual feedback in RViz via MoveIt! Motion Planning plug-in (exemplary planning scene: figure 2.3). Debugging is an easy task, as well.

**Disadvantages** The central assumption necessary for this method is that the obstacles and their positions are fixed after receiving a grasping command. In practice and at this stage in the ReGround project however this is not a major limiting factor as the usage is still rather artificial. Furthermore, this method relies on very accurate size estimates by the anchoring pipeline. The vision sensor has some weaknesses: The estimate of bounding box size of objects fluctuates and deteriorates with increasing distance from the sensor. This is due to the infrared depth sensing technology used by the Kinect.

**Result** Despite these disadvantages, this method works very well in practice. Once all obstacles are added to the planning scene, MoveIt! reliably finds paths to grasp or drop without colliding with the objects on the working surface. As mentioned in the Pose Generator, the grasping pose height is additionally adjusted in accordance with the bounding box size generated by the anchoring system. In some instances however, if the object to be gripped is close to `base_link`, i.e. farther away from the Kinect sensor, the gripper grasps above the object because the size estimation from the anchoring system is more unreliable with increasing distance from the sensor.

**Future Considerations** To tackle the problem of inadequate size estimation with increasing distance from the depth sensor, a correction of the bounding box estimate according to its distance to the anchoring system could be a valid solution. This correction values are inherent to the sensor which means they will have to calculated by hand and trial and error. Additionally, a sensor that is more precise should eliminate this problem all together.

While the Planning Scene was filled with obstacles manually in the first approach, the second approach revolved around automatic obstacle avoidance with an Octomap, a map that is created from a 3D point cloud and *"explicitly represents not only occupied space, but also free and unknown areas"* (Hornung et al., 2013). Embedding

this type of map[4] in a MoveIt! Planning Scene provides significant constraints to the MoveIt! Planner and can be done continuously and automatically from raw sensor data.

**Advantages** Using an Octomap generated from near real-time point cloud data from the Kinect depth sensor provides a continuous and accurate planning space for MoveIt!. The Planning Scene generated from the Octomap with raw sensor data provides greater detail than the MoveIt! Planning Scene generated from information garnered from the Anchoring System for multiple reasons: For one, the Planning Scene accuracy depends on bounding boxes generated by the Anchoring System with inherent size estimation discrepancies. Furthermore, the system sometimes struggles to recognize smaller objects, e.g. the toy car seen in figure 4.2.

Furthermore, this method interfaces nicely with the previously discussed method by removing Octomap elements that are both occupied by Planning Scene objects like the table top or bounding boxes of anchored objects, which are manually added via the Planning Scene API, and an Octomap element itself. We are still left with the benefits of the Octomap for moving targets. This allows for easy integration in future work as it wouldn't require a change to existing manual Planning Scene code.

**Disadvantages** As a user who depends on visualizations in RViz to evaluate gripping and dropping motions, I experienced the Octomap to be a significant bottleneck for RViz performance. This was surprising as the map capturing frame rate was set to a conservative 10Hz, as well as having access to a more than adequate computer with an Intel i7 CPU and an NVidia GTX 970 GPU. The CUDA driver was working as expected and performing well under other circumstances, e.g. during anchoring which is run in large parts on the GPU. However, the system monitor did not show any abnormalities while being under this supposed heavy computational load. This issue will have to be investigated further in order to allow the permanent use of this method.

[4]Octomap Planning Scene Integration: https://archive.fo/GFGgh (archived 30.04.2019)

**Result** The Kinect and its ROS driver[5] provide three levels of point cloud accuracy. Namely *sd*, *qhd*, and *hd*, with *hd* being the most precise. While the Octomap generation was successful and well integrated into the MoveIt! planning scene, I was having trouble with point cloud noise around the Jaco grippers. In particular, noise accumulated right behind the gripper from the Kinects perspective. This was surprising as MoveIt! filters the point cloud before building an Octomap. Furthermore MoveIt! was able to find a solution for most gripping tests, which it was unfortunately unable to execute due planning costs being exorbitantly high. It is unclear why this is the case and requires further testing.

**Future Considerations** There are multiple adjustments that could be experimented with to make planning with the Octomap in place possible

(a) *Filtering*: point cloud filtering is handled by MoveIt! by default. There is no way to turn this feature off, but handing MoveIt! a pre-filtered point cloud may lead to better results. However, this may filter too many details that are necessary for an accurate Octomap.

(b) *Kinect calibration*: While unlikely in my opinion, it may be necessary to re-calibrate the Kinect sensor[6]. The Anchoring System works well and precise however, which supports the assumption that the depth sensor is correctly calibrated.

(c) *Mounting or model discrepancy*: Even though poses can be moved to with high reliability, there may be a slight measurement error in the URDF model of the robot that causes the model seen in RViz to be slightly off. On one hand, this in turn could explain the noise above the gripper, as the URDF model in RViz might be shown too low. On the other hand, a significant model discrepancy will and should result in high gripping failure rates, the opposite of which was confirmed in chapter 5.

[5]ROS Kinect Driver: https://github.com/code-iai/iai_kinect2 (visited 29.04.2019)
[6]Kinect Calibration: https://github.com/code-iai/iai_kinect2/tree/master/kinect2_calibration (visited 29.04.2019)

## 4.4 ROSSpeechPublisher

The Language Grounding System supports linguistic input in the form of text. This is perfectly fine for testing and the ReGround pipeline, though work has been started on an iOS application[7] that records speech and publishes its text equivalent generated by the NSSpeechRecognizer library[8], once completed. At the time of this writing recording and recognizing speech is implemented, while errors persist when trying to publish the translated speech.

[7] ROSSpeechPublisher: https://github.com/juliangaal/ROSSpeechPublisher (visited 30.04.2019)
[8] NSSpeechRecognizer NLP library
https://developer.apple.com/documentation/appkit/nsspeechrecognizer (visited 12.04.2019)

# Chapter 5

# Testing and Results

Multiple tests have been conducted to verify gripping performance, planning performance and general reliability. However, before discussing the test methodology and results I have to reiterate the scope of this paper. The ReGround Project and the necessary robot manipulation are still in very stages (as of 11/2018) and a lot of hardware and software changes need to be made to turn it into a production quality project. Because of this, the focus of my work did not delve deep into different planning algorithms and planning optimization but focused on basic manipulation and reliability. Nevertheless, test with various levels of complications were executed as part of the gripping and manipulation tests.

As mentioned in section 3.3.2, one grasp pose as part of a *GraspPoseGoal* includes the move to a pose generated by the Pose Generator, the grasp itself, and a short lift off the table surface, while the *DropPoseGoal* simply executes a drop pose and drops the gripped object at the specified position.

## 5.1 Background

The most error-prone link in the ReGround Pipeline (figure 4.1) turned out to be the language processing system, as it was still being actively developed at the time of testing (October 2018). As mentioned in chapter 2, the revised system will be presented at SpLU-RoboNLP 2019[1] on 06.08.2019. If you are interesting in video recordings of a number of tests with the preliminary linguistic interface in use, please refer to Appendix C.

[1] SpLU-RoboNLP 2019: https://web.archive.org/web/20190430085238/https://splu-robonlp.github.io/ (archived 30.04.2019)

To be able to judge the robustness of the anchoring system, its size estimation and the viability of my contribution to ReGround however, the language processing part was skipped for the tests in this chapter. Additionally, using the robot with the linguistic interface in place significantly increases testing times and decreases the sample numbers in the given time frame.

## 5.2 Recording

All tests were recorded both in video as well as text, the latter of which was enabled by extending the Jaco Manipulation ROS Package[2] described in chapter 3 to publish status messages. This was accomplished by utilizing ROSs ability to publish custom messages. In this case, the following log information was published during all tests for each goal that was sent to the ROS Action Server.

```
string timestamp
MoveIt! configuration
        string planner
        float32 planning_time
        bool allow_replanning
        int32 planning_attempts
MoveItGoal goal
        string description
        Pose current_pose
        Pose target_pose
        Bounding Box bounding_box
Planning Scene planning_scene
        uint64 number_of_obstacles
        Bounding Box Array obstacles
string result
```

This published data was then recorded by a ROS subscriber and saved as a `.csv` file to allow for plotting and other analysis.

## 5.3 Analysis

The recorded `.csv` files were analyzed with Pythons `pandas`[3] and `matplotlib`[4] packages and were of the following format:

[2]Jaco Manipulation: https://github.com/juliangaal/jaco_manipulation (visited 10.05.2019)
[3]Pandas: https://archive.is/58LoU (archived 30.04.2019)
[4]Matplotlib: https://archive.fo/D7nTO (archived 30.04.2019)

```
Time;Type;CurrentPose;TargetPose;Result
Sat Sep 29 14:37:14 2018;grasp_pose;(0.458984672944,-0.345...;success
Sat Sep 29 14:37:22 2018;drop_pose;(0.20588313455,-033475....;success
Sat Sep 29 14:37:40 2018;grasp_pose;(0.205830785417,0.00212..;success
Sat Sep 29 14:38:07 2018;drop_pose;(0.353035882155,-0.6052...;failure
Sat Sep 29 14:38:18 2018;grasp_pose;(0.523391611605,-0.532...;success
```

Relevant for the analysis were the following parameters (first line in csv file):

**Time** Used to determine average gripping time

**Type** Many goals are sent to the ROS Action Server, not all were recorded however. Important types for the analysis were `grasp_pose`, a pose that was meant to grasp an object, and `drop_pose`, a pose meant to drop already grasped objects.

**Current Pose** The initial pose determined by the MoveIt! Framework at the time of requested execution.

**Target Pose** The requested pose.

**Result** The gripping result (gripped, not gripped) had to be recorded by hand. The gripper used did not provide pressure sensors to determine the success or failure of a grip, and designing a feedback loop with the anchoring system in place turned out to be very difficult to achieve for numerous reasons, including the fact that the anchoring system loses sight of the gripped object once the gripper is closed. More on these issues will be discussed in 6.

## 5.4 Result Types

I differentiate between 3 possible result types for each grip

**Success** Gripper functions as expected and is able to grasp the target object. This result type will be colored **green** in the following plots.

**Failure** Gripper is unable to grasp the target object. Either, measurement and accuracy errors in the anchoring system, which result in slight miscalculations in the Pose Generator, are severe enough to, in some cases, make the gripper touch and move the object away from the target position while executing the computed trajectory. Or, planning errors by MoveIt! that indicate a calculated

collision. There are ways to tackle this issue, and some are discussed in 5.7. This result type will be colored **red** in the following plots.

**Inconclusive** Gripper did function as expected and is able to grasp the target object, but, e.g., *collides* (slightly) with the target object, leaving its position *unchanged* nevertheless. This is also recorded when the grip strength is not adequate for the object and is dropped mid-trajectory. While some may argue to interpret this as a failure as well, this separate result type was very important for parameter tuning and its effects done in section 5.7. This result type will be colored **orange** in the following plots.

## 5.5 Baseline Test

**Objectives** Testing the reliability of gripping and planning *without* the vision or anchoring system in place. It was chosen to provide a sense of the overall state of the system and to allow optimization before moving on to more complex tests involving existing components of the ReGround project.

**Procedure** Grasping and dropping locations were randomly generated from a Gaussian distribution across the working surface. The poses were executed one by one, and a ball with an estimated bounding box volume of 343 $cm^3$ with a height, width and depth of 7.0 cm each was placed on the work surface just before the gripper reached the randomly generated grasp pose. This specific test was executed twice, *continuously* with 50 random poses for gripping and dropping each.

**Result** The Baseline Test was a success. The ball was gripped in every single instance, and no planning errors were present. It is important to point out that this was a very synthetic process, nevertheless enabling the quick progression to more challenging tests.

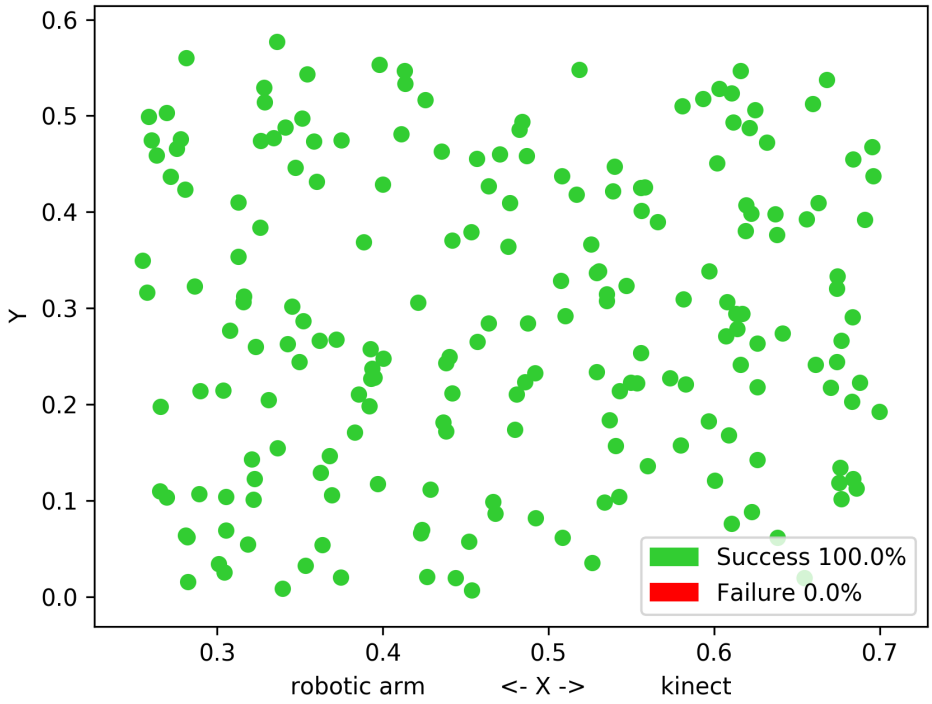



**Figure 5.1:** Baseline Test results: executed poses and result types from top view. Only grasp poses are shown

## 5.6 Anchoring Test

**Objectives** Testing the reliability of gripping and planning *with* the vision or anchoring system in place. The communication between the anchoring system with its detected objects and the robotic arm is central to the project, and needed to be tested extensively.

**Procedure** In this test only drop positions were randomly generated from a Gaussian distribution across the working surface, as the grasping poses were determined by the Pose Generator, after receiving a centroid and dimension estimate of the target object from the anchoring system. The procedure is best described in the following pseudo code:

```
WHILE number of actions is smaller than N
DO
 GET centroid and size estimate of target from anchoring system
 GENERATE grasp pose from centroid and size estimate
 EXECUTE generated grasp pose
 EXECUTE random drop pose
```

```
 MOVE to resting position "home"
 INCREMENT number of actions
END
```

**Result** While this test performed 9% worse than the Baseline test, it did provide the first quantifiable data necessary to interpret the state of a large part of the ReGround pipeline. Of 100 grasp poses determined by the Anchoring System and Pose Generator, 91% were indeed successful, which provides a satisfying initial benchmark.

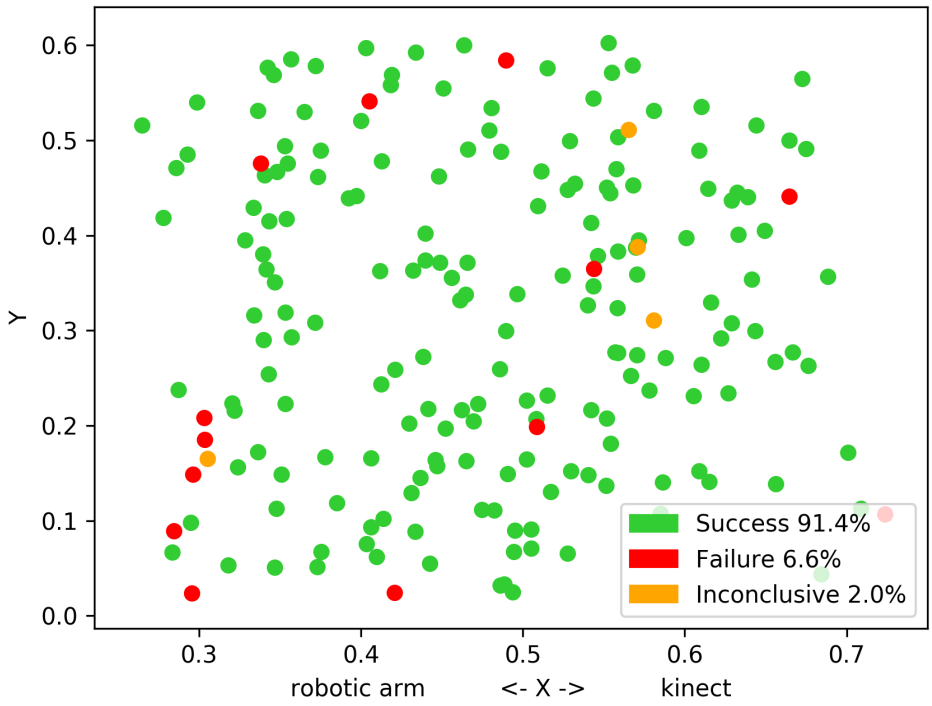



**Figure 5.2:** executed poses and result types from top view

### 5.6.1 Results Explored

By default, the MoveIt! Framework uses a planner configuration called *RRTConnectkConfigDefault*. It enables the use of a popular planner called *RRTConnect*, implemented as part of the Open Motion Planning Library (OMPL)[5]. While this planner is nice to start working on projects, it is characterized by fast, but sub-optimal planned trajectories. This is due to the fact that it implements *"two rapidly-exploring random*

[5]OMPL: https://archive.fo/124dz (archived 30.04.2019)

*trees (RRTs) rooted at the start and the goal configurations. The trees each explore space around them and also advance towards each other through, the use of a simple greedy heuristic"* (Kuffner and LaValle, 2000). While MoveIt! allows further parameterization in its configuration file[6], it is ultimately limited by the main pitfall of greedy heuristic search, namely non-guaranteed optimality.

Additionally, inaccuracies in centroid and size estimation by the anchoring system increase with the distance to the vision sensor. For objects located close to Jaco's base, or on the opposite end of the workspace from the Kinects point of view, the error in centroid estimation was approximately +2 cm along the x-axis. Errors in size estimation and its implications will be discussed in section 5.8.

Combining both of these factors explains all errors seen in figure 5.2. While work has been started to support grasping poses other than from straight above the object, the current system support a fully functional top grasp. The robotic arm - *under the assumption of an* ***optimal*** *trajectory* - has much more room for error in the top grasp scenario. The grippers are able to grasp the target object as long as it is in reaching distance of a single finger. This means that the centroid estimate can be inaccurate by about approximately $\pm 3$ cm on the x-axis, or $\pm 1.5$ cm on the y-axis, demonstrated in figure 5.3. While this may result in a light touch of the target object, it will still be gripped.

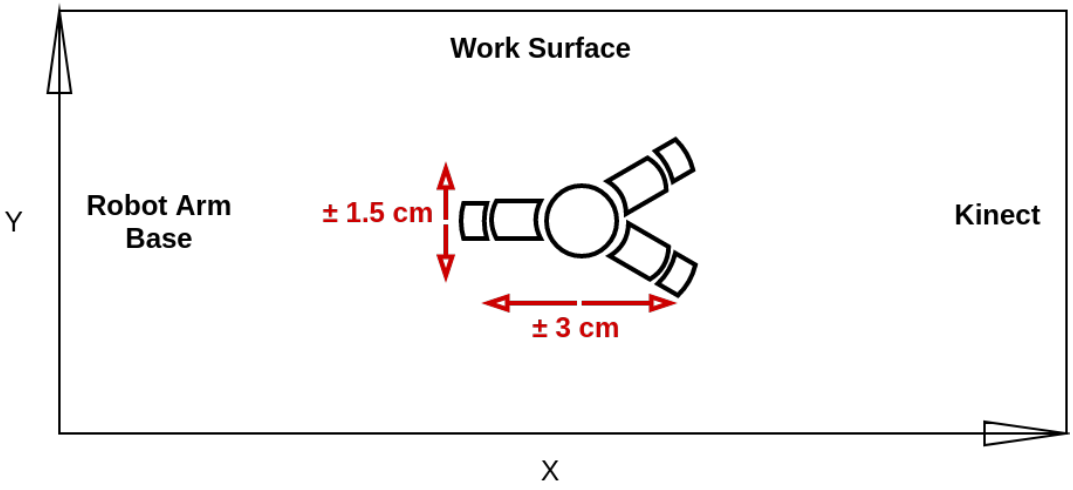



**Figure 5.3:** Gripper in top grasp position from birds eye view. Top grasp error range explained. Scaling adjusted for clarity.

[6]OMPL MoveIt! configuration https://github.com/juliangaal/wpi_jaco/blob/develop/jaco_moveit_config/config/ompl_planning.yaml#L27 (visited 12.04.2019)

In all cases of grasping errors MoveIt! calculated a ***sub-optimal*** trajectory, however, resulting in a collision with the target object due to its path, often coming in sideways. In that process, the target was moved from the original target position and the grip failed, as described in figure 5.4. While these errors where frustrating at times, they were clearly observable, traceable, and ruled out the possibility of any other mistakes in the Jaco Manipulation wrapper around the MoveIt! planner API, which indeed was encouraging. A suggested technique to mitigate this problem is

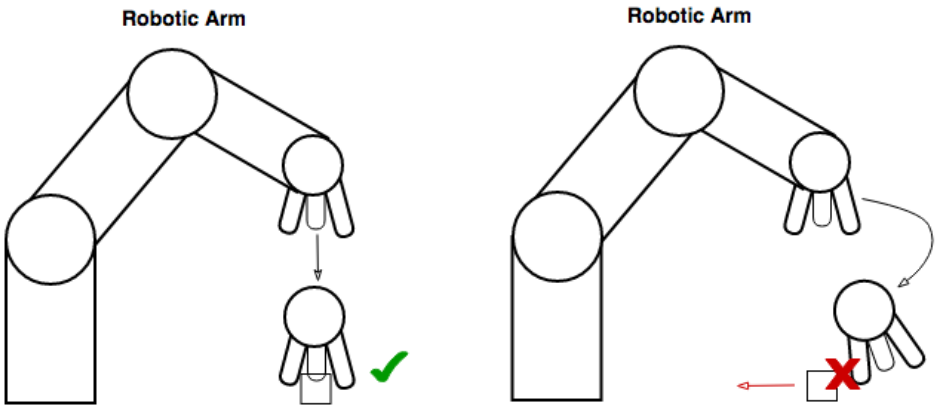



**Figure 5.4:** Optimal path and exemplary sub-optimal path and its effect on gripping

discussed in section 6.2.1.

## 5.7 Improvements

The next iteration of tests was designed to tune existing parameters. Therefore, the failed grasp poses from the Anchoring Test in the previous section were picked specifically marked on the work surface and repeated for each test. While the results are not directly comparable to due to differing sample sizes, they offer great insights into the parameters that have positive effects on grasp poses with high failure rates (figure 5.2).

**Planner and Planning Time** Reiterating on the parameters mentioned in 5.2, `allow_looking`, `allow_replanning` and `num_planning_attempts` had no effect on gripping results.

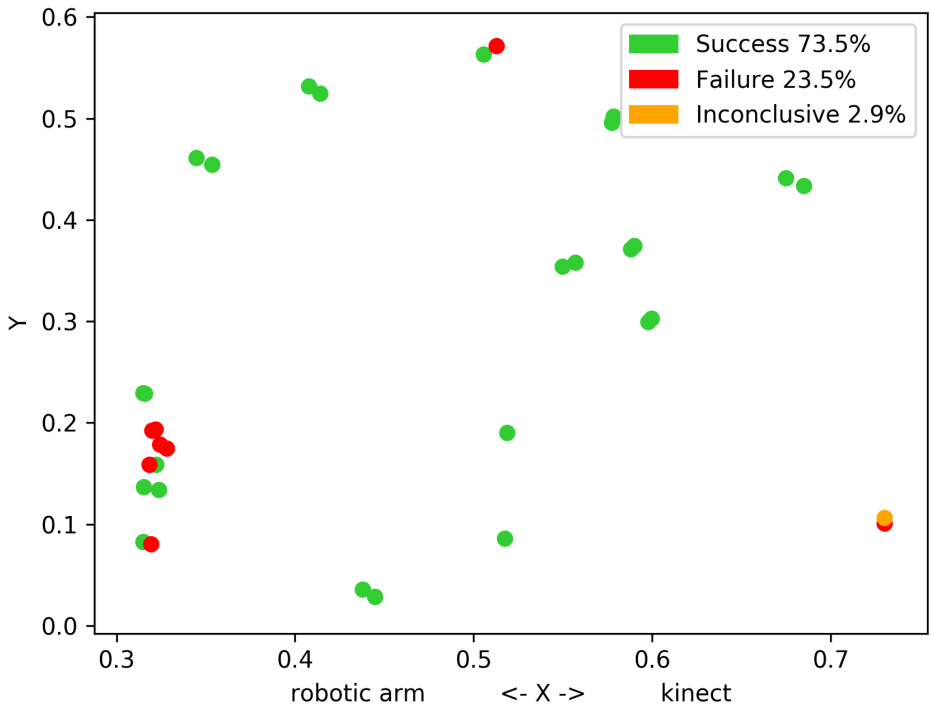



**Figure 5.5:** Planner RRTConnect with previous failed poses and a planning time of 1.5 seconds

This may be the case because of the relative simplicity of the ReGround environment and its largely obstacle free workspace. However, changes to the parameters `planner` and `planning_time` had significant effects and on the overall performance.

It is important to note that the RRTConnect plug-in for MoveIt! does not, in fact, improve upon valid trajectories, once a solution is found. While this contributes significantly to a perceived responsiveness of the robot to commands, different planners that improve on already found results are also available as MoveIt! plug-ins. One of those, *RRTstarkConfigDefault*, implements an improvement of RRTConnect, called *RRT**, that improves an already computed path *"by removing 'redundant' edges, i.e., edges that are not part of a shortest path from the root of the tree. [...] this amounts to a 'rewiring' of the RRT tree, ensuring that vertices are reached through a minimum-cost path"* (Karaman and Frazzoli, 2011). The following section will be devoted to tests with RRT* path planning and various planning times, in which only previously failed or Inconclusive poses shown in figure 5.2 were repeated.

In all cases tested, a solution was found by RRTConnect in significantly less time than 1.5 seconds. Figure 5.5 perfectly demonstrates the issue of planning with RRTConnect, as many of the previously failed grasps in fact did not fail in this round of testing. While an initial solution is found by RRT* in under 1.5 seconds, improvements to this initial path cannot take advantage of the residual planning time reliably, as shown in figure 5.6, with an only modest improvement to the base test with RRTConnect and a planning time of 1.5 seconds.

Figure 5.6 to 5.9 show the same previously failed grasp positions with different planning times, in comparison. It was frustrating to see the necessity for a significantly increased planning time. Even with a planning time of 9 seconds, not all poses were grasped. While the test performance did increase with a total success rate improvement of 12%, the responsiveness of the entire system did take a significant hit. A future kitchen robot will have to be very responsive in order to be useful and productive.

**Implementing a Pre-Grasp Pose** A common method to reduce sub-optimal planned trajectories is to generate a pre-grasp pose, as pose that moves the gripper just in front or over the target object, depending on the orientation, before executing the remaining trajectory to the target object. As the distance between the pre-grasp pose and target pose is very slim, planners are much more likely to find an optimal solution.

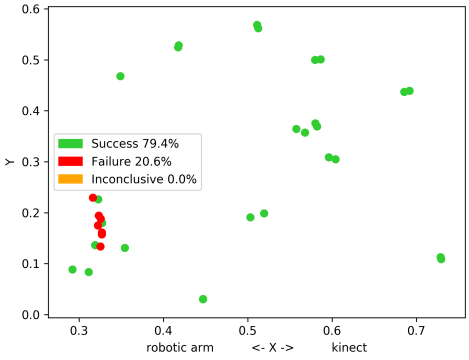



**Figure 5.6:** Planner RRT* and a planning time of 1.5 seconds

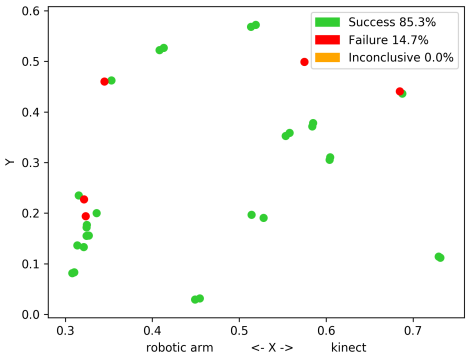



**Figure 5.7:** Planner RRT* and a planning time of 3 seconds

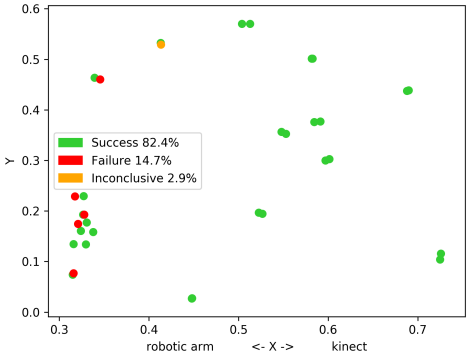



**Figure 5.8:** Planner RRT* and a planning time of 6 seconds

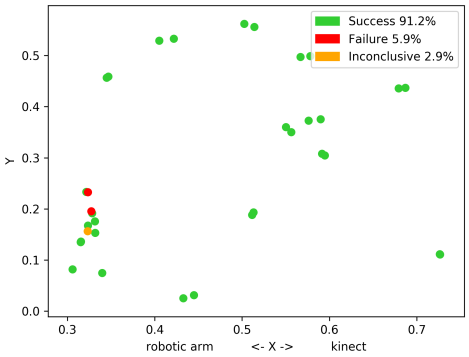



**Figure 5.9:** Planner RRT* and a planning time of 9 seconds

For efficiency, the pre-grasp pose is set by applying an height offset to the already generated grasp pose by the Pose Generator discussed in chapter 3.2, on the Jaco Action Server itself. This means that the pose generation and sending to the Action Server has to happen only once, instead of for each pose in a grasp motion. However, for more complicated poses like front grasps or side grasps, it will be necessary to integrate a distinct pre-pose into it.

The results of using this method were very promising, as it resulted in the highest success rate of any anchoring tests. It is important to note that it was possible to keep the original, "bad" planner, RRTConnect, while achieving this. Because the gripper can't touch the target object in its pre-grasp pose due to its height, the planned trajectory to this point - no matter how sub-optimal - has no effect on the target object. The original anchoring test (RRT*, 1.5 seconds planning time) was rerun with this method in place (figure 5.10). Grasping errors where almost entirely eliminated,

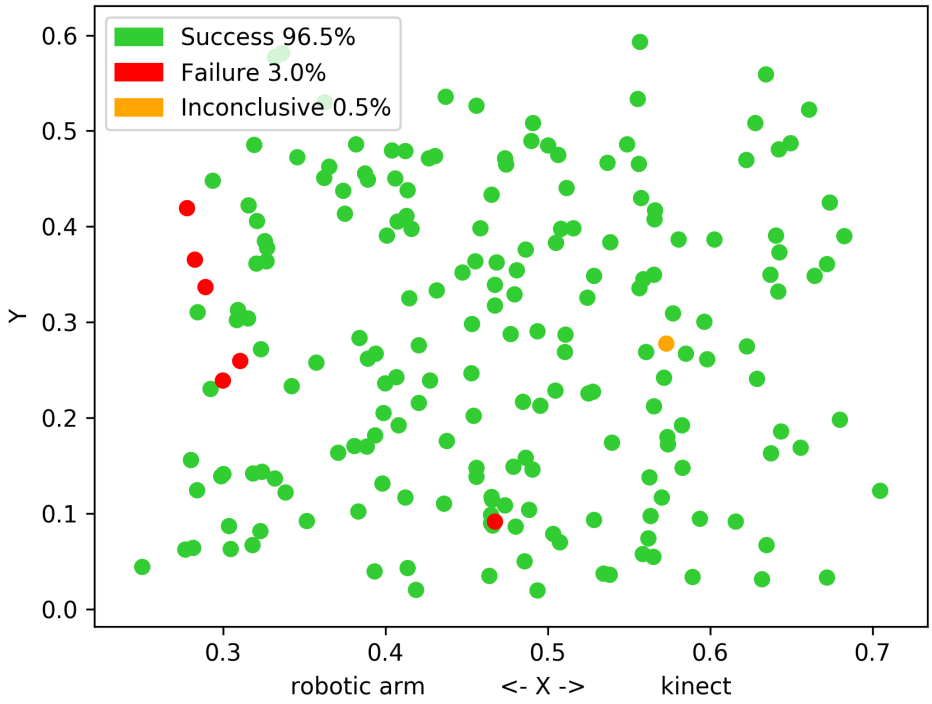



**Figure 5.10:** Original Anchoring Test (figure 5.2) with pre-grasp pose lowest planning time of 1.5 seconds

with a success rate of 96.5% in a total of 100 grasps. The mistake described figure 5.4 did unfortunately still occur, even though the planner had to only adjust the pose by 10 cm. While this took place in only 3% of all grasps, I am confident that - after comparing the results in figure 5.6 and 5.7 - running the same test with RRT* as the planner and a planning time of greater than or equal 3 seconds will result in a success rate of 99% or higher..

## 5.8 Obstacle/Occlusion Anchoring

**Intention** The last test was implemented to see if surface occlusion had any effect of our gripping results. Unfortunately, this was tested before the pre-grasp pose was part of the software stack. Therefore, error rates similar to results described in section 5.6 are to be expected.

**Procedure** The procedure was separated into 3 different levels of surface occlusion: 11%, 22%, and 33%. This may seem low, but in a realistic environment, 33% surface occlusion implies very little space for anything else (see figure 5.11).

For each surface occlusion rate, 20 grasp tests were performed and recorded. This number is very low compared to previous tests, but they were adequate to reason about the results and were not meant to be used as a further benchmark. During the test, the object was placed in the workspace covered with obstacles, followed by a grasp attempt calculated by the Anchoring-Jaco pipeline, and repeated. It is important to note that objects were placed in a positions know to have average success rates, based on previous results to eliminate factors that could negatively influence planning.

**Figure 5.11:** Exemplary surface occlusion for obstacle anchoring test: 33%. The white line marks the border of the work space.

**Result** While error rates and gripper behavior matched expectations, this was very hard to test, for one reason in particular: because of the position of the anchoring system and its angle onto the work surface, objects are often times estimated to be longer than they actually are. Because all recognized objects on the work surface are part of the MoveIt! planning scene, these size discrepancies can lead to planning errors as the planner detects a collision in the scene, even though there is no contact between objects on the *actual* surface.

However, with these instances removed, the success rate reached 91% across all recorded data points.

Comparing individual occlusions against each other would require significantly more data points. The fact that most failed grasps occurred with 22% occlusion could be a random occurrence when taking the (at this stage) RRT-Connect planner and low sample size into account. That said, it was important and reassuring to observe that not a single collision took place between actual objects in the scene and the gripper.

## 5.9 Reproducibility

A guide to reproduce all the tests explained in this chapter, including the analysis, is presented in appendix B. Furthermore, the code and recorded .csv files are also available online[7]. It is provided as a ROS package and requires the full installation of all elements[8] of the ReGround pipeline.

[7] Test suite: https://github.com/juliangaal/jaco_manipulation_test (visited 20.04.2019)

[8] ReGround Workspace installation helper: https://github.com/juliangaal/reground_workspace (visited 15.04.2019)

# Chapter 6

# Conclusions

## 6.1 Summary of Achievements

Over the course of three months (August - October 2019) I went from building a general purpose interface to the robotic arm, Jaco, to integrating all existing software components developed by members of the pre-existing ReGround working group. The resulting system took novel approaches in *relational* symbol grounding and successfully applied it to a kitchen robot that is able to perform object manipulation given linguistic commands. Video recordings of the system can be found in appendix C.

The Anchoring System working together with Jaco Manipulation enables grasp/-drop success rates of up to 95.6%, as proven in extensive tests in chapter 5. Jaco Manipulation Server has a stable API and will need little to no modification in future iterations of the project. Furthermore, Jaco Manipulation Client as a library itself has also reached a mature state and can be used in conjunction with the Arm Service to easily extend the ReGround logic layer (table 4.1).

Lastly, I am confident that the test suite developed forms a solid base for future development of the system and benchmarks with it. It will significantly speed up the development of improvements to the system as it allows for more confident programming and validation of experiments.

## 6.2 Discussion and Future Improvements

Due to my limited time with the physical robot, I would like to explore ideas that developed towards the end of my work in the AASS Lab[1], after completing the multitude of tests explained in chapter 5. They provide a head start for future participants of the ReGround team or people who wish to continue work on the Jaco Manipulation package[2] itself.

### 6.2.1 Artificial Constraints and Flexible Planning

While the Pre-Grasp Pose implemented and discussed in section 5.7 enabled significant improvements in grasping performance and enabled the use of the faster planner, RRTConnect, MoveIt! offers the ability to set artificial path constraints[3]:

1. Position constraint[4]
2. Orientation constraint[5]: Fix, e.g., orientation of `jaco_link_hand`
3. Joint constraint[6]

When setting constraints, a weight, or importance, has to be given to each constraint and allows fine tuning. A position constraint could be implemented to move from the resting position (see figure 3.1 to the pre-grasp pose, while both an orientation constraint and position constraint could be applied to move from the pre-grasp pose to the target pose, supposedly eliminating the error case demonstrated in figure 5.3. With these constraints in place, it would be interesting to repeat the tests with multiple planning times and *RRT** from section 5.7.

Further optimization could be attempted by using different planners and planning times for different phases of the motion. For example, *RRTConnect* could be used to plan the trajectory to the pre-grasp pose, while the more important step, the final grasping motion, could be planned with RRT* and an elongated planning time. The

[1] AASS lab: http://archive.is/mrP49 (archived 01.05.2019)
[2] Jaco Manipulation: https://github.com/juliangaal/jaco_manipulation (visited 01.10.2019)
[3] Path constraints: http://archive.is/i07YI (archived 01.05.2019)
[4] Position constraint: http://archive.is/RSadA (archived 01.05.2019)
[5] Orientation constraint: http://archive.is/zHH9O (archived 01.05.2019)
[6] Joint constraint: http://archive.is/so44X (archived 01.05.2019)

experiments in section 5.7 to judge by, a planning time of > 3 seconds for RRT* seems reasonable.

### 6.2.2 Grasp Pose Generator

When extending the capabilities of the Grasp Pose Generator discussed in chapter 3.2 it should be the top priority to implement front grasp poses with different orientations to enable, e.g. a grasp from the front, slight left and slight right of the object. While this missing feature does not entail a significant limitation for the current state of the ReGround project, it might be necessary to implement for more sophisticated actions like pouring liquid.

With increasing complexity of the Grasp Pose Generator it makes sense not to generate a single pose and execute it, but to generate multiple poses at once. In case one generated grasp pose can't be reached due to constraints in the arm or environment, a fallback pose in the list of possible poses could be attempted next. This would be a more robust solution in more restricted or complex environments.

### 6.2.3 Octomap Obstacle Avoidance

Detailed in section 4.3.2, it is unclear why all attempts at using raw 3d sensor data from the Microsoft Kinect to create a live Octomap have failed. Regardless, Octomap is the superior obstacle avoidance technique and should be attempted to integrate into the ReGround Pipeline for optimal obstacle avoidance in future experiments with the robotic arm.

### 6.2.4 Hardware

Lastly, I would like to mention possible hardware changes that could benefit the project greatly. Even though these changes are very unlikely, they are fun to think about. Recent advances in pressure sensitive robots with so called *"soft grippers"* (Homberg et al., 2015) have allowed for great successes when handling a multitude of object types with different densities and shapes while simultaneously recognizing the grasped object. This could eliminate the need for manual adjustments of the orientation generated for the top grasp pose and, most importantly, provide feedback about the status of the grasp attempt. Due to the pressure sensitivity of the

gripper, it would be possible to detect a failed grasping attempt, and either repeat it with a greater force or different angle. Furthermore, it could be used to validate the image classification done by the Anchoring System which could, in turn, making communication with the robot less error prone.

# Bibliography


[1] Laura Antanas et al. "Relational symbol grounding through affordance learning: An overview of the ReGround project". In: *International Workshop on Grounding Language Understanding (GLU)*. Satellite of INTERSPEECH. 2017. URL: http://www.speech.kth.se/glu2017/papers/GLU2017_paper_3.pdf.

[2] Ozan Arkan Can et al. *Learning from Implicit Information in Natural Language Instructions for Robotic Manipulations*. 2019. arXiv: 1904.13324 [cs.AI].

[3] Silvia Coradeschi and Alessandro Saffiotti. "An introduction to the anchoring problem". In: *Robotics and Autonomous Systems* 43 (2003), pp. 85–96. DOI: 10.1016/S0921-8890(03)00021-6.

[4] B. S. Homberg et al. "Haptic identification of objects using a modular soft robotic gripper". In: *Proc of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 1698–1705. DOI: 10.1109/IROS.2015.7353596.

[5] Armin Hornung et al. "OctoMap: an efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous Robots* 34.3 (2013), pp. 189–206. ISSN: 1573-7527. DOI: 10.1007/s10514-012-9321-0.

[6] Sertac Karaman and Emilio Frazzoli. *Sampling-based Algorithms for Optimal Motion Planning*. 2011. arXiv: 1105.1186 [cs.RO].

[7] J. J. Kuffner and S. M. LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Proceedings of the IEEE International Conference on Robotics and Automation*. 2000, 995–1001 vol.2. DOI: 10.1109/ROBOT.2000.844730.

[8] Andreas Persson et al. *Semantic Relational Object Tracking*. 2019. arXiv: 1902.09937 [cs.RO].

# Appendix A

# Detailed Installation Instructions

The installation instructions assume a working ROS installation and a ROS compatible operating system. If you are not familiar with ROS and how to set up a ROS workspace, please refer to the official ROS Tutorials[1].

First, let's start with a list of required packages (easy download method mentioned shortly):

1. Jaco manipulation

   The ROS Package for all available Jaco Manipulation[2] must be saved in the existing ROS workspace. It depends on a fork of the original Kinova API that is compatible with ROS Kinetic, the WPI Jaco driver wpi_jaco[3] and necessary robot state message types rail_manipulation_msgs[4] are also required.

2. ReGround workspace

   In addition to all packages required for Jaco's manipulation, there are ROS Packages for Anchoring[5], the ReGround Brain[6] as well as Language Grounding[7] available.

[1] ROS Tutorials: http://archive.fo/PVjZP (archived 01.05.2019)
[2] Jaco Manipulation: https://github.com/juliangaal/jaco_manipulation (visited 01.05.2019)
[3] WPI Jaco driver: https://github.com/juliangaal/wpi_jaco (visited 01.05.2019)
[4] Rail Manipulation Msgs: https://github.com/juliangaal/rail_manipulation_msgs (visited 01.05.2019)
[5] https://bitbucket.org/reground/anchoring/src/master/ (visited 01.05.2019)
[6] ReGround Brain: https://bitbucket.org/reground/reground_brain/src/master/ (visited 01.05.2019)
[7] Language System: https://bitbucket.org/reground/language/src/master/ (visited 01.05.2019)

3. Tests

   Large efforts have been made to make all tests described in chapter 5 reproducible and extensible. Therefore, it has been separated from the Jaco Manipulation ROS package from above. This adds the flexibility to improve the existing arm manipulation code, add more tests without affection the arm manipulation, or both. A ROS package for the existing tests, jaco_manipulation_test[8], is provided as well.

The easiest way to download the entire ReGround software stack is with ws_tool[9] by cloning this git repository[10] and running the following command in the folders root:

```
$ wstool update -t src
```

After this, follow the installation instructions in the ROS packages *README* files.

[8] Jaco Manipulation Test suite: https://github.com/juliangaal/jaco_manipulation_test (visited 20.04.2019)

[9] ws_tool: https://archive.fo/n4kyi (archived 12.05/2019)

[10] Download help: https://github.com/juliangaal/reground_workspace (visited 22.04.2019)

# Appendix B

# Reproducing Tests

Please follow the installation instructions in appendix A before continuing. For some cases, multiple ROS nodes have to be started. While this could be made a little easier with launch files, they create large amounts of output in a single terminal window. Therefore, it is advised to start some nodes/launch files individually, as done so below.

First, let me describe how to perform the tests mentioned in chapter 5. Later, I will explain how to generate plots from the collected data points. Note that all packages mentioned in this chapter are already linked in appendix A. `node_root` refers to the root of the Jaco Manipulation Test package in your Catkin Workspace. Please remember: You need to record the gripping result manually.

## B.1 Anchoring Test

**Use Case** continuous test of grasping object detected by Anchoring System. An object is gripped 100 times and dropped off at a random position immediately. After placing the target object onto the work surface for the very first time, the system works autonomously.

**Requirements** :

- generate random drop poses in `<node_root>/post/generate` with

```
$ cd <node_root>/post/generate
$ python generate_anchoring_poses.py
```

  Default: 100 drop poses. Change it in the file if needed

- Jaco Manipulation needs to be running:

```
$ roslaunch jaco_manipulation jaco_manipulation.launch
```

- launch Anchoring System by typing

```
$ roslaunch anchoring anchoring_static_tf.launch
```

- Run: launch Anchoring Test and recorder

```
$ roslaunch jaco_manipulation_test anchoring_test.launch
```

  The recorder started in this launch file will listen to messages sent by Jaco Manipulation and record them in

  `<node_root>/post/anchoring_test_recording.csv`

The implementation of this test can be found in the anchoring_test.cpp[1] file.

## B.2 Baseline Test

- Use Case: continuous test of grasping object *without the vision system in place* An object is gripped and dropped off at random position for a total of 50 runs. Grasping and dropping alternate. The object has to be placed at the randomly generated target position *by hand*.

- Requirements: Jaco Manipulation needs to be running:

```
$ roslaunch jaco_manipulation jaco_manipulation.launch
```

  If you want RViz visualization, use parameter `use_rviz:=true/false`

- Run: launch anchoring test and recorder in the Jaco Manipulation package

```
$ roslaunch jaco_manipulation_test baseline_test.launch
```

  The recorder will listen to messages sent by Jaco Manipulation and record them in `<node_root>/post/baseline_test_recording.csv`

The implementation of this test can be found in the `obstacle_test.cpp`[2] file.

[1] Anchoring Test file: https://github.com/juliangaal/jaco_manipulation_test/blob/master/src/anchoring_test.cpp (visited 01.15.2019)

[2] Baseline Test file: https://github.com/juliangaal/jaco_manipulation_test/blob/master/src/baseline_test.cpp (visited 01.15.2019)

## B.3 Obstacle/Occlusion Test

- Use Case: continuous test of grasping object in between obstacles on the work surface. An object is recognized, gripped, dropped immediately afterwards on the same spot, for a total of 30 times. Grasping and dropping alternate. After each drop, the test pauses for 10 seconds, in which the object has to be manually placed to a different location.

- Requirements: Jaco Manipulation needs to be running:

```
$ roslaunch jaco_manipulation jaco_manipulation.launch
```

  If you want RViz visualization, use parameter `use_rviz:=true/false`

- Run: launch anchoring test and recorder in the `jaco_manipulation` package

```
$ roslaunch jaco_manipulation_test baseline_test.launch
```

  The recorder will listen to messages sent by Jaco Manipulation and record them in `<node_root>/post/baseline_test_recording.csv`

The implementation of this test can be found in the `obstacle_anchoring_test.cpp`[3] file.

Next, I will walk you through the process of generating the plots. For each test type, there is a corresponding file in `<node_root>/post/analyse/`, e.g. `anchoring_recordings.txt` for the Anchoring Test. In this, you have to add the file path of your recordings, after moving it from `<node_root>/post/<type>_recording.csv` to a new folder of your choosing in `<node_root>/post/results`. Lastly, you need to adjust the values of column *Gripped* in the recorded .csv files: by default, the value is *Default*, but needs to adjusted to either "success", "failure", or "inconclusive", according to your hand written recordings of the gripping performance. You are now ready to generate plots: run

```
$ python analyse_recordings.py
```

[3] Obstacle anchoring file: https://github.com/juliangaal/jaco_manipulation_test/blob/master/src/obstacle_anchoring_test.cpp (visited 01.15.2019)

in `<node_root>/post/analyse/`. Please note that only python 2.7 is supported, due to compatibility issues between python 3 and ROS kinetic.

In some instances however, it may be helpful to test single/a few single arm motions during the development of Jaco Manipulation Client or Server. These "quick tests", so to speak, were used for quick evaluation of code changes. Therefore, and other organizational reasons, the following files are part of the original Jaco Manipulation Package.

## B.4 client_grasp_test.cpp

- Use Case: provides example interaction with Jaco Manipulation Client to test predefined grasp and drop poses and bounding boxes
- Requirements: Jaco Manipulation from the Jaco Manipulation needs to be running:

  ```
  $ roslaunch jaco_manipulation jaco_manipulation.launch
  ```

  If you desire RViz visualization, use parameter `use_rviz:=true/false`
- Adjust and Compile: Adjust file `test/client_grasp_test.cpp`
- Run:

  ```
  $ rosrun jaco_manipulation client_grasp_test
  ```

## B.5 client_test.cpp

- Use Case: provides simple example interface to Jaco arm to test predefined poses (edit in file), joint states and goals defined in MoveIt config
- Requirements: Jaco Manipulation needs to be running:

  ```
  $ roslaunch jaco_manipulation jaco_manipulation.launch
  ```

  If you desire RViz visualization, use parameter `use_rviz:=true/false`
- Adjust and compile: Adjust file `test/client_test.cpp`

- Run:

```
$ roslaunch jaco_manipulation client_test
```

- Evaluation: As everything is hardcoded, simply compare the movement of the arm with the desired movement

# Appendix C

# Video Recordings

The following links are video recordings of the Robotic Arm and ReGround components in action. They demonstrate pick and place actions with the arm, show the natural language input in a text interface and visualize the recognized information by the Anchoring System.

1. https://vimeo.com/335614665
2. https://vimeo.com/335614736
3. https://vimeo.com/335614582
4. https://vimeo.com/335614429
5. https://vimeo.com/335614337
6. https://vimeo.com/335614948