ARTIFICIAL INTELLIGENCE VISION ROBOTIC ARM

A THESIS SUBMITTED TO
THE UNDERGRADUATE SCHOOL OF ELECTRIC-ELECTRONIC
ENGINEERING
OF
ISTANBUL TECHNICAL UNIVERSITY

BY

SAMER ALDABBAS & MOUSTAFA HASSANEIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF BACHELOR OF SCIENCE
IN
CONTROL & AUTOMATION ENGINEERING

JUNE 2022

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

The world is changing and fourth industrial revolution has been started which transforms the way of work and living for human-being. At the heart of this fourth revolution is artificial intelligence which is the ability of machines to match and perhaps one day surpass the cognitive ability of their human creators. These are early days in the brave new world of artificial intelligence but the potential benefits are vast (Mahtani et al., 2018).

The creation and implementation of a remote laboratory in the field of autonomous and robotic engineering is described in this work. The laboratory is built on an instructional platform centered on a robotic arm that allows engineering students to simulate a practical laboratory by comparing their theoretical conclusions to the robot's real-world trajectories. As a result, the study reported in this article focuses on the adaption of robotics and control ideas in a remote laboratory. Moreover, this paper aims to find an optimal design in terms of electrical, control, and mechanical schemes; as it is aimed to find the most efficient human-robotic assistant but all trials will be performed in laboratories to test the product from vast angles (Causo et al., 2018).

Nowadays, the pharmaceutical sector is a big consumer of laboratory robots, and it's possible to claim that the history of lab automation mirrors the evolution of contemporary drug discovery in this business (Choi et al., 2011). The pharmaceutical sector was experiencing a time of great profitability in the late 1980s and early 1990s, but pharmaceutical corporations were in dire need of labor-saving technology as the scope and complexity of drug research and development grew substantially. As mentioned by Robert Bogue: "The Tox21 collaboration will transform our understanding of toxicology with the ability to test in a day what would take one year for a person to

do by hand". Therefore, one ought to invest in remote robotic arm in medical/chemical fields (Wise et al., n.d.).

1.1 Motivation for enforcement of Robotic Arms in Laboratories

In recent years, a large variety of apps based on distance–learning platforms have been developed. They have shown tremendous promise in terms of improving training in the disciplines of autonomous and robotic engineering. The goal of this application under the offered platform is to develop and create a remote control application for a laboratory manipulator that connects to the Internet. To do this, we began with an educational application that simulates a realistic laboratory in which engineering students may verify theoretical conclusions by comparing them to real-world routes followed by the robotic arm (Aaltonen & Barth, 2005).

Figure 1.1: Robotic Application in Laboratories

Knowing how to operate a robotic agent is becoming more common as robots becomes more integrated into many aspects of human life (Jara et al., 2008). Robotic toys are quite widespread nowadays; you can find everything from assistive robots to aid persons with disabilities with feeding to assistant robots for physical activity for adolescents, all of which are part of the vast array of innovations found in the state of the art. Until they reach essential support points as teleoperations intermediates, robotic agents express themselves in a variety of ways (Bogue, 2012). This sort of application allows users to observe the presence of robots in man's activities and their impact on his quality of life. As a result, developing platforms for robot control training that are adaptable to a wide variety of ages is critical for their effective usage and development. With the ability to control various systems remotely, such as air conditioning, robot control, and similar systems, as well as the advantages of virtual laboratories, remote laboratories for telecontrol robots now provide solutions for needs such as training in various areas, such as in the case of teleoperation (Kumar et al., 2018).

Figure 4.1 shows one design that are currently used in pharmaceutical industry.

1.2 Model Proposal & Mission

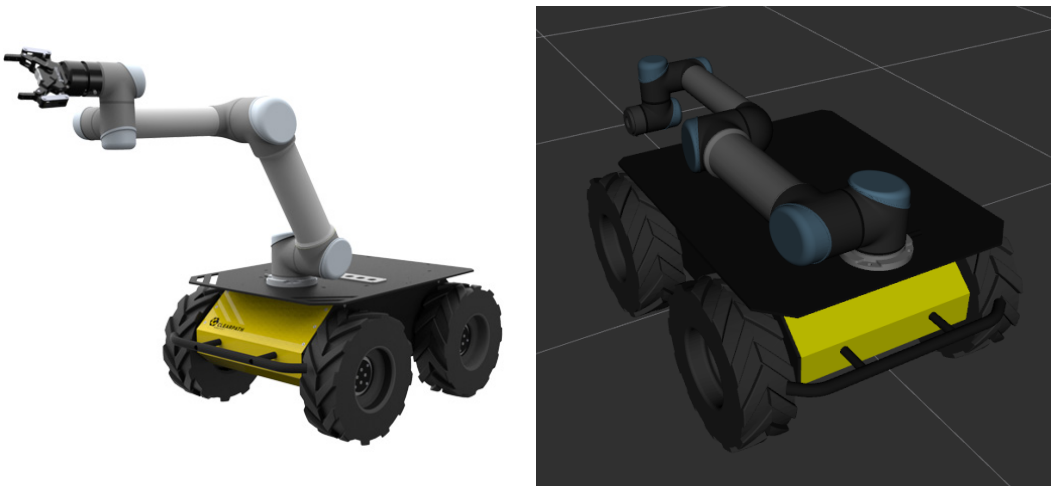

Figure 1.2: 3D Model of Mobile Manipulator

In this study, we provide an unique ROS-based mobile manipulation framework. As mobile manipulation mostly occurs in unstructured situations, an on-board RGBD camera is implemented as a visual perception system. The system is meant to isolate manipulation and navigation as two distinct goals, allowing for its easy transfer from simulation training to real-world testing. After a thorough simulation review, the robot manipulation strategy should be evaluated on a real mobile manipulation robot in order to grab a variety of item kinds from random starting positions. This framework is, to the best of our knowledge, the optimum mobile manipulation model successfully implemented on a real-world robot. The mission is to design and code a ROS package that is ready to be open sourced by any entity and be mount on various kinds of mobile robots.

Figure 1.3: Real-world Application Using UR5e & Husky Config.

CHAPTER 2

HARDWARE ARCHITECTURE

The hybrid mobile manipulator is discussed in this chapter under five distinct branches. These are the serial manipulator, the mobile platform, the gripper, the camera, and the main computer.

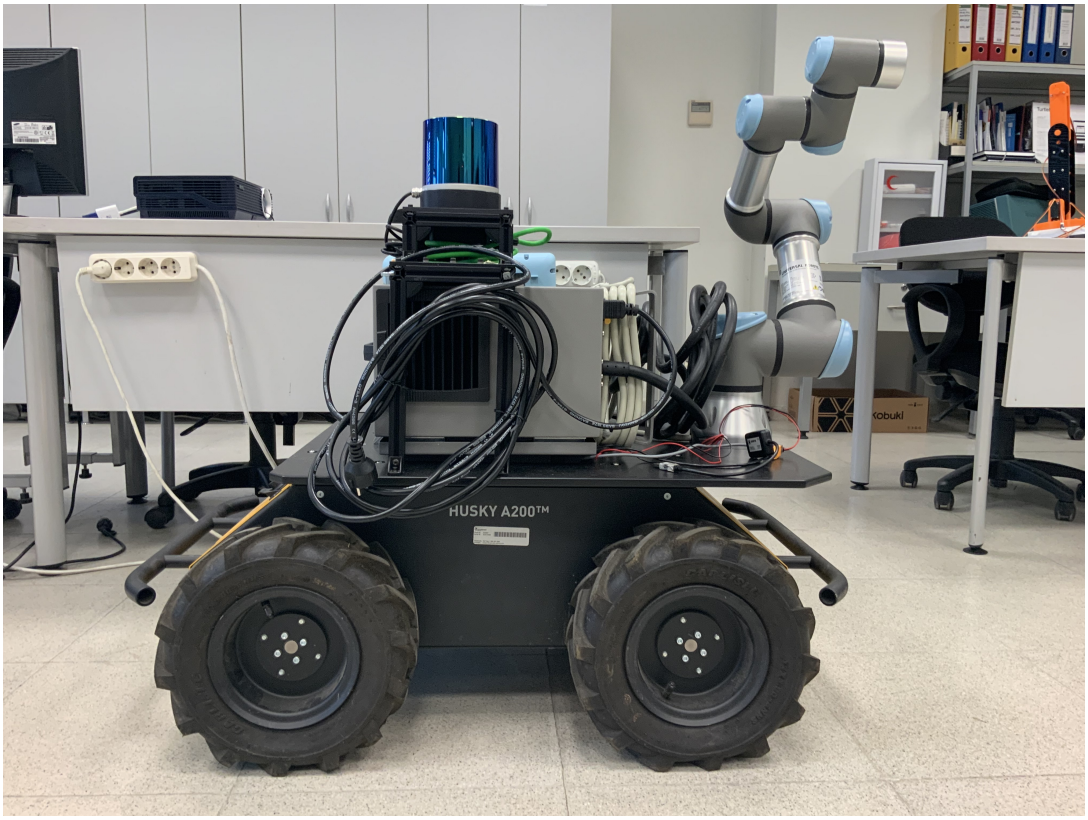

Figure 2.1: Husky & UR3 Configuration

2.1 Technical Specifications of the UNIVERSAL ROBOT UR3

Six-axis arms constitute the majority of industrial robot arms seen in modern production environments. For six-axis robots, robot technology is always evolving. Six-axis robots provide several advantages, including enhanced wrist action and flexibility, software and programming capabilities, and a variety of mounting possibilities. There is also an extensive selection of robot sizes, payloads, and speeds. Additionally, six-axis robots are becoming easier to operate, repair, and replace. The four most common manufacturers of industrial robots are FANUC, Motoman, KUKA, and ABB. They have created and produced big and small six-axis industrial robot arms for industrial applications. There are a variety of clean room and food grade robots, in addition to other robots designed for severe work environments, to prevent worker harm or hazard pay. Due to the mentioned arguments, a 6 Dof manipulator is selected. For the robot selection, The Universal Robots UR3 is the company's newest collaborative tabletop robot designed for light assembly chores and automated workbench applications. The compact tabletop robot weighs only 11 kilograms (24.3 pounds), but the arm has six axes, a payload of 3 kilograms (6.6 pounds), 360-degree rotation on all wrist joints, and unlimited rotation on the end joint. The UR3 collaboration robot is the market's most adaptable collaborative tabletop robot for working alongside humans. It is a small, compact robot used for medium to long-term installations requiring repeatability. Its modest size makes it ideal for installing on or within a machine. It is ideal for applications demanding uniform product quality involving assembly, polishing, gluing, and screws. It can be utilized at a separate work station mounted on a table to pick, assemble, and place parts in efficient production flows, and due to its compact design and simple programming, it has a low total cost of ownership and an extremely quick payback period. The UR3 robot is utilized in numerous industries, including medical devices, circuit boards, and electronic components. It is constructed with the same collaborative technology as the UR5 and UR10 robot arms, which has been proven effective. All three collaborative robots feature the same safety system and repeatability of 0.1 mm. The UR3 robot has automated jobs up to 3 kg (6.6 lb) (as its name suggests), has a reach radius of up to 500 mm (19.7 in), and offers the industry's quickest payback. The UR3 robot is the optimal solution for businesses seeking to automate 6-axis applications where size, safety, and

affordability are crucial. It contains a 6-axis articulated arm that can do a variety of operations in close collaboration with human employees, including pick-and-place, soldering, screwing, gluing, and painting. The UR3 robot has a small footprint and is simple to set up and configure. As a result, it is simple to swap activities to fulfill a variety of manufacturing requirements, making it the ideal assistance for assembly applications demanding uniform product quality. The UR3 robot is used for picking and positioning parts in optimized production processes. The UR3 robot is simple to use, quick to place in a machine, faster, more rigidly constructed, and has an established user interface and workflow. It has a large ecosystem of tools, integrators, programs, and use cases and is less expensive. Cobots are developed for collaborative applications; they have varying specifications; they can increase production, prevent injuries, and boost morale; and they are designed to facilitate collaboration. Cobot arms are utilized by businesses of all sizes. They are simple to operate, safe, and may work alongside people without the need for safety precautions. They are easy to instruct, and persons with little programming experience may quickly program them to accomplish tasks because to the 3D interface's clarity. They are intelligent and not merely a collection of gadgets programmed to repeat tasks without monitoring potential changes or interferences in their immediate surroundings.

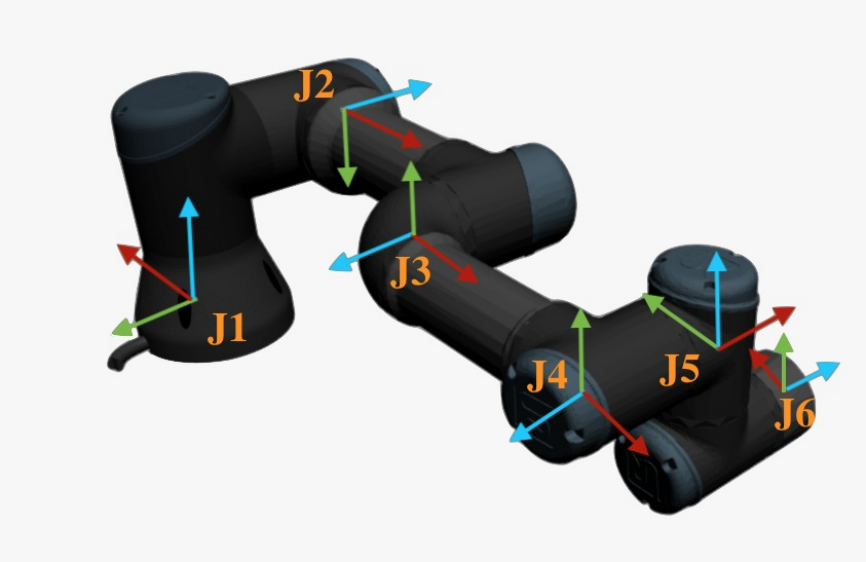

Figure 2.2: UR3 Manipulator

Robot Specifications		Robot Motion Speed		Robot Motion Range
Axis	6	J_1	$180°/s(3.14rad/s)$	$\pm 360°$
Payload	3 kg	J_2	$180°/s(3.14rad/s)$	$\pm 360°$
H-reach	500 mm	J_3	$180°/s(3.14rad/s)$	$\pm 360°$
Repeatability	$\pm 0.1mm$	J_4	$360°/s(6.28rad/s)$	$\pm 360°$
Robot Mass	11 kg	J_5	$360°/s(6.28rad/s)$	$\pm 360°$
Structure	Articulated	J_6	$360°/s(6.28rad/s)$	$\pm 360°$
Mounting	Table, Floor, Inverted, Angle			

Table 2.1: Technical Specifications

2.2 Technical Specifications of the Husky UGV

Husky was the ROS-enabled platform for field robots that supports ROS with software packages. The hybrid robot's mobile platform is also seen in Figure 2.3. It has encoders with a high resolution that are essential for state estimation. Additionally, it can hold up to 75 kg. Husky would also serve as a power source for the serial manipulator, microcontroller, and other sensors. Simultaneously, with the manipulator package, Husky is marketed with a Universal Robots UR5 robot, however in the ITU Robotic Laboratory, it is configured with a UR3 robot.

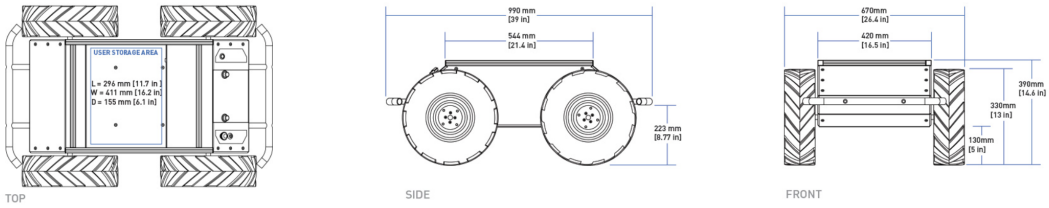

Figure 2.3: Husky UGV Dimensions

2.3 Gripper Selection

For the gripper selection, a gripper compatible with the UR3 manipulator is selected due to difficulties to install or design a custom gripper. After intensive research, the 2-Finger Adaptive Gripper is selected as an ideal gripper for the aim of this project. The 2-Finger Gripper is available in two configurations: 85 mm (2-Finger 85) or 140 mm (2-Finger 140). For the simulation of this project, 2-Finger 85 has been selected.

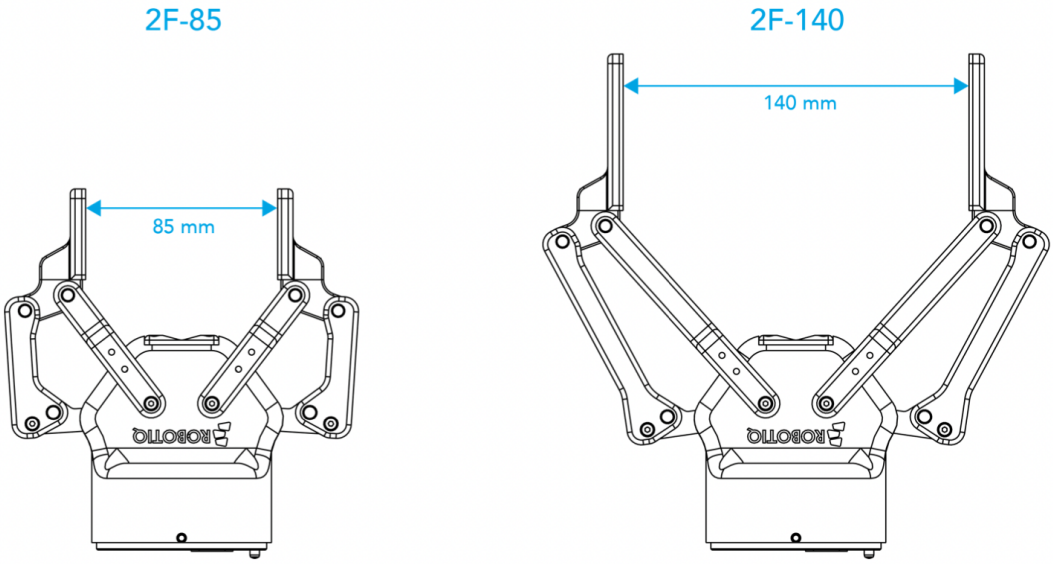

Figure 2.4: The 2-Finger 85 and 140 mm versions

As depicted in the figure below, the 2-Finger Gripper features two articulated fingers, each with two joints (two phalanxes per finger). The grasp-type gripper may make contact with an object at up to five locations (two on each of the phalanges plus the palm). The fingers are under-actuated, which means that they contain fewer motors than joints. This arrangement enables the fingers to automatically conform to the shape of the object they are grasping, hence simplifying gripper control.The gripper is directly powered and controlled by a single device cable with a 24V DC supply and Modbus RTU communication over RS-485. Refer to the Electrical Setup section for details on wiring, and the Control section for gripper control (various software packages are available for control via various robot controllers).

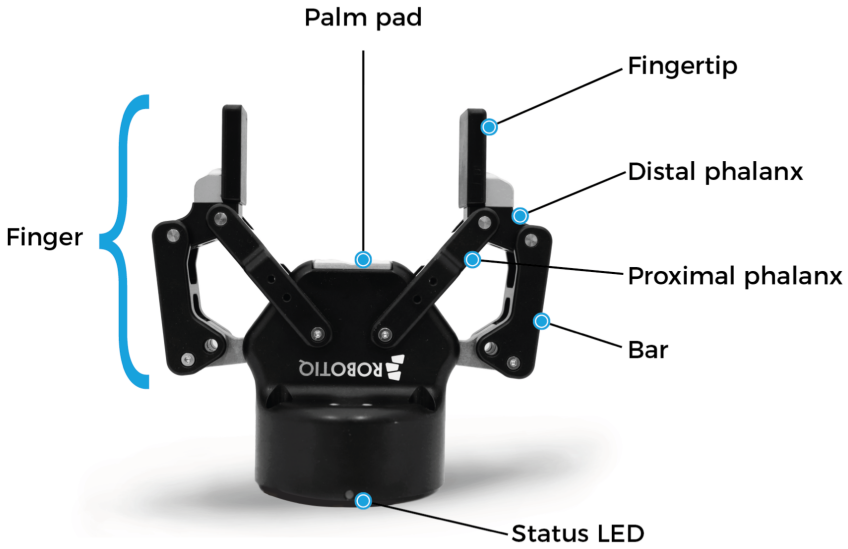

Figure 2.5: Robotiq 2-Finger Adaptive Gripper

The 2-Finger includes an integrated object detecting feature that use indirect sensing techniques. With a simple object detection bit, the gripper status will indicate if an object is picked or not when picking an object using the "go to" command (0 or 1). After detecting an object, the gripper will stop. If the object is dropped, the gripper will automatically close to retain it until the object is identified or the "go to" command's position objective is reached.

Specification	2-FINGER 85	
Gripper Opening	85 mm	3.35 in
Minimum object diameter (for encompassing)	43 mm	1.69 in
Maximum height	162.8 mm	6.4 in
Maximum width	148.6 mm	5.85 in
Weight	925 g	2.04lbs
Grasp Force	20 to 235 N	4.5 to 52.8lbf
Finger speed	20to150 mm/s	0.8 to 5.9 in/s
Position repeatability 1	0.05 mm	0.002 in
Force repeatability	$+/-10\%$	0.016 in

Table 2.2: Technical Specifications of the Robotiq 2-Finger Adaptive Gripper

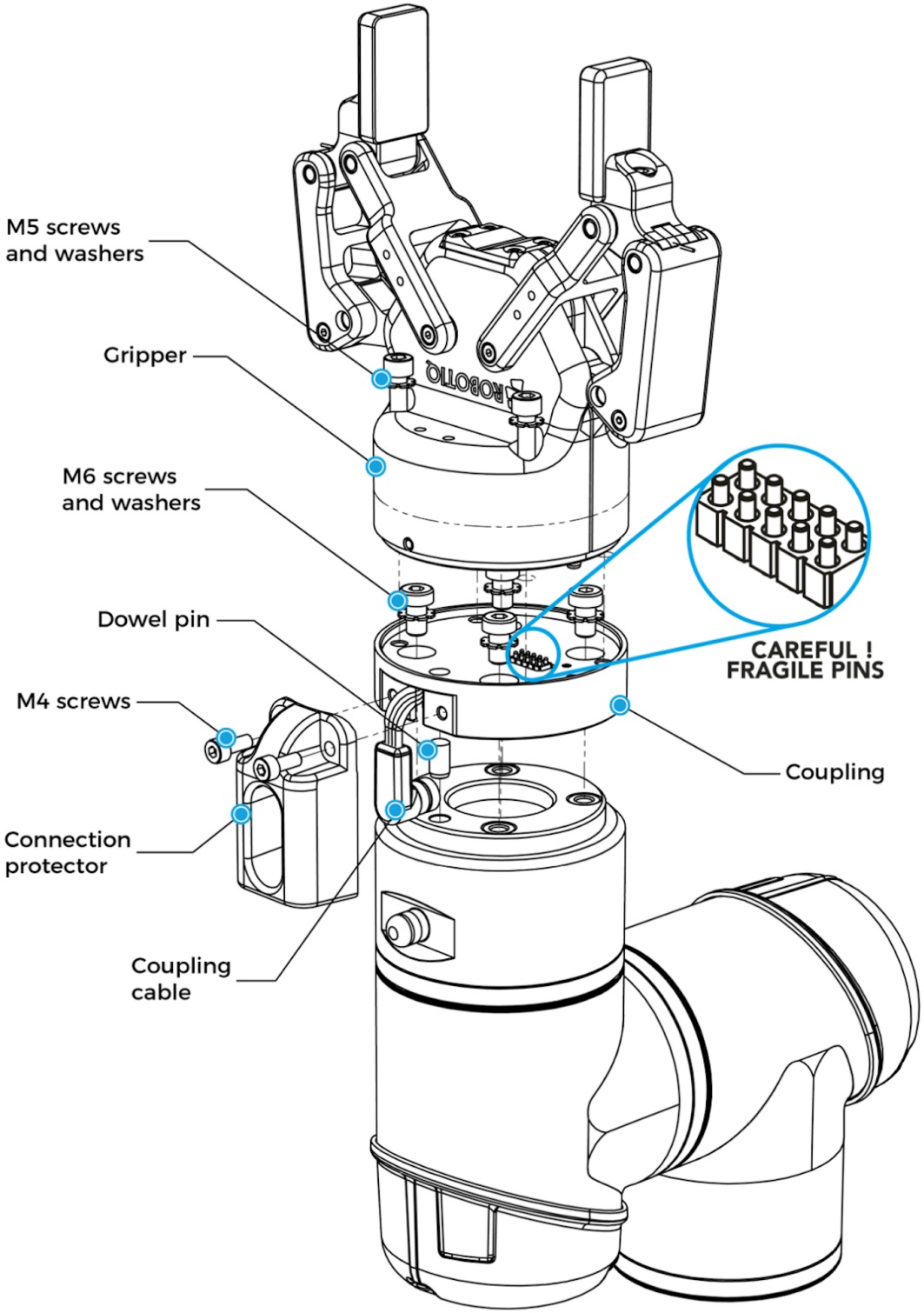

Figure 2.6: Installing the gripper onto the robot wrist for e-Series

2.4 Camera Selection

Figure 2.7: Robotiq 2-Finger Adaptive Gripper

For the RGBD camera selection, a strong and reliable solo 3D camera, Astra features the Orbbec 3D microprocessor with VGA color. Astra was designed to be extremely compatible with existing OpenNI applications, making it an ideal 3D camera for OpenNI-based apps already in existence. Astra has a range of 0.4 to 2 meters. The detailed specifications are given in the table below.

Model N°	ORBBEC ASTRA PRO
Size	$160 \times 30 \times 40(\ \mathrm{mm})$
Weight	$300\ \mathrm{g}$
Range	$0.4 - 8\ \mathrm{m}$
Depth Image Size	$720\mathrm{p} \quad 30\mathrm{FPS}$
Field of View	$60°$ horiz. $\times 49.5°$ vert. (73 diagonal)
Microphones	2
Operating Systems	Windows, Linux, Android

Table 2.3: Astra Camera Technical Specifications

2.5 NVDIA Jetson TX2 Developer Kit

The Jetson TX2 is the most potent and power-efficient embedded artificial intelligence computing device. This supercomputer-on-a-module consuming 7.5 watts enables true AI computing at the edge. It features an NVIDIA Pascal^{TM}-family GPU and 8GB of RAM with a memory bandwidth of 59.7GB/s. It features a number of common hardware connections that make it easy to integrate into a variety of devices and form factors.

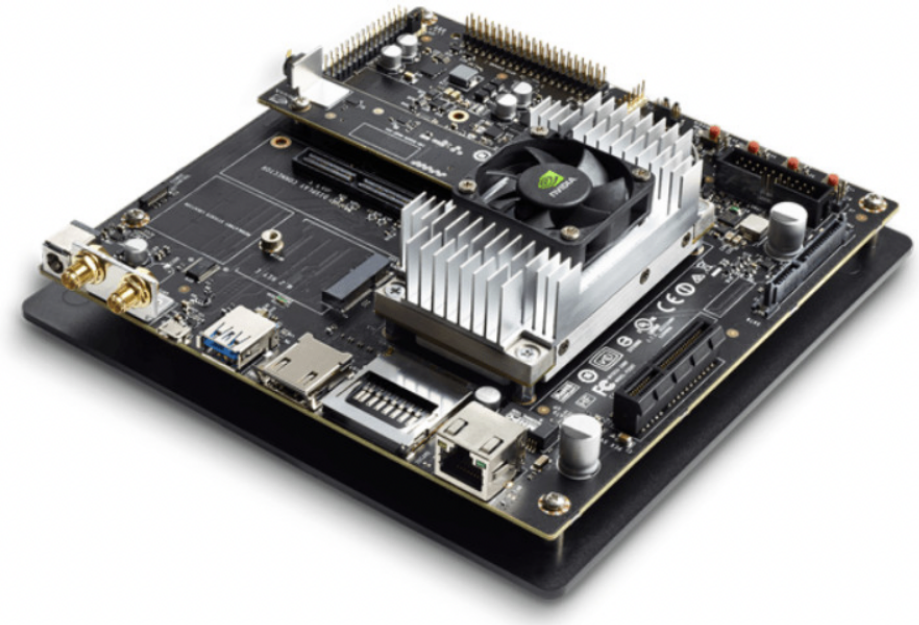

Figure 2.8: NVDIA Jetson TX2 Developer Kit

CHAPTER 3

KINEMATIC ANALYSIS OF THE SERIAL MANIPULATOR

3.1 Forward Kinematic

Kinematic modelling based on the Denavit-Hartenberg (D-H) parameters (Denavit & Hartenberg, 1955).The DH parameters are a four-parameter minimum representation of the robot kinematics $(\mathrm{a_i}, \mathrm{d_i}, \alpha_\mathrm{i}, \theta_\mathrm{i})$. Figure 3.1 shows the UR robot design and its free body diagram (FBD) with connecting reference frames. Table 3.1 shows the DH parameters. These four parameters can specify the transformation from coordinate frame link i+1 to frame i+1 in terms of four elementary rotations and translations. initially a rotation about the z axis by θ_i, followed by a_i translation about the z axis by d_i, then a translation about the x axis by a, and finally a rotation about the x axis by α_i. The matrix A represents this transformation (Whitty, 2012).

(Collet et al., 2009)

$$ {}^{\mathrm{i-1}}_{\mathrm{i}}\mathrm{A}\left(\mathrm{a_i}, \theta_\mathrm{i}, \mathrm{d_i}, \alpha_\mathrm{i}\right) = \mathrm{Rz}\left(\theta_\mathrm{i}\right)\mathrm{Tz}\left(\mathrm{d_i}\right)\mathrm{Tx}\left(\mathrm{a_i}\right)\mathrm{Rx}\left(\alpha_\mathrm{i}\right) \tag{3.1} $$

$$ \begin{aligned} {}^{\mathrm{i}}_{\mathrm{i-1}}\mathrm{A} &= \begin{bmatrix} \mathrm{C}\theta_\mathrm{i} & -\mathrm{S}\theta_\mathrm{i} & 0 & 0 \\ \mathrm{S}\theta_\mathrm{i} & \mathrm{C}\theta_\mathrm{i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathrm{d_i} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \alpha_\mathrm{i} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \mathrm{C}\alpha_\mathrm{i} & -\mathrm{S}\alpha_\mathrm{i} & 0 \\ 0 & \mathrm{S}\alpha_\mathrm{i} & \mathrm{C}\alpha_\mathrm{i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ {}^{\mathrm{i}}_{\mathrm{i-1}}\mathrm{A} &= \begin{bmatrix} \mathrm{C}\theta_\mathrm{i} & -C\alpha_\mathrm{i}\mathrm{S}\theta_\mathrm{i} & \mathrm{S}\alpha_\mathrm{i}\mathrm{S}\theta_\mathrm{i} & \mathrm{a_i}\mathrm{C}\theta_\mathrm{i} \\ \mathrm{S}\theta_\mathrm{i} & \mathrm{C}\alpha_\mathrm{i}\mathrm{C}\theta_\mathrm{i} & -\mathrm{S}\alpha_\mathrm{i}\mathrm{C}\theta_\mathrm{i} & \mathrm{a_i}\mathrm{S}\theta_\mathrm{i} \\ 0 & \mathrm{S}\alpha_\mathrm{i} & \mathrm{C}\alpha_\mathrm{i} & \mathrm{d_i} \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{3.2} $$

The transformation matrix for the End Effector with respect to the base frame is then derived by multiplying the transformation matrices for each successive joint, from joint 1 to joint n (n = 6 in our application). As a function of joint position, the transformation matrix ${}^{0}_{6}\mathrm{A}$ is

$$ {}^{0}_{6}\mathrm{A}\left(q_1,\ldots,q_6\right) = \prod_{i=1}^{6} {}^{i}_{i-1}\mathrm{A} \equiv \begin{bmatrix} \mathrm{R}_{3\times3} & \mathrm{P}_{3\times1} \\ 0 & 1 \end{bmatrix} \tag{3.3} $$

The aforementioned equations should yield a forward kinematics model of the robot.

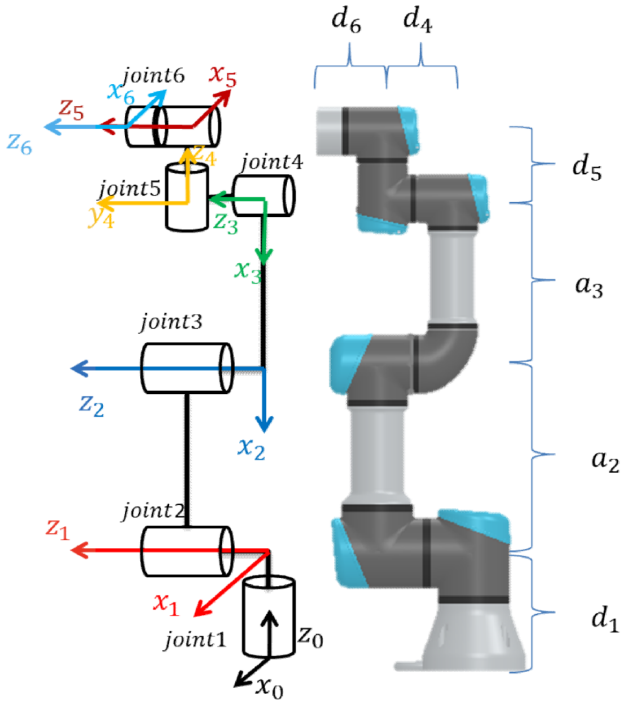

Figure 3.1: UR Robot FBD with links reference frames

joint	θ_i(rad)	a_i (m)	d_i (m)	α_i (rad)
1	θ_1	0	d_1	$\pi/2$
2	θ_2	a_2	0	0
3	θ_3	a_3	0	0
4	θ_4	0	d_4	$\pi/2$
5	θ_5	0	d_5	$-\pi/2$
6	θ_6	0	0	0

Table 3.1: UR3 Robot DH parameters

3.2 Inverse Kinematic Solvers

In this section two inverse kinematic solvers is discussed. These are track_ik, and IKFast.

3.2.1 IKFast

IKFast, the Robot Kinematics Compiler, is a strong inverse kinematics solver supplied within Rosen Diankov's OpenRAVE motion planning program. Unlike other inverse kinematics solvers, IKFast can analytically solve the kinematics equations of any complicated kinematics chain, and output language-specific files (like C++) for subsequent usage. The ultimate result is incredibly reliable solutions that can execute as quickly as 5 microseconds on modern processors

3.2.2 track_ik

TRAC-IK offers an alternate Inverse Kinematics solution to KDL's well-known inverse Jacobian approaches. Specifically, KDL's convergence algorithms are based on Newton's technique, which is ineffective in the presence of joint limitations – a regular occurrence in robotic platforms. TRAC-IK operates two IK implementations simultaneously. One is a straightforward modification to KDL's Newton-based con-

vergence method that detects and mitigates local minima caused by random jumps at joint limits. The second is a SQP (Sequential Quadratic Programming) nonlinear optimization strategy that employs quasi-Newton techniques that handle joint limits more effectively. By default, the IK search returns instantly upon convergence of either of these methods. In order to return the "optimal" IK solution, secondary constraints of distance and manipulability must also be specified. In this project, track_ik is selected as a solver in moveit_group.

CHAPTER 4

AUTONOMOUS MISSION SOFTWARE & SIMULATION

4.1 Codes

We will be using the simple grasping ROS package for this unit. This package was created by Mike Ferguson, so kudos to his excellent effort. For this course, we have modified the original code in order to better accommodate the subject matter, however you can view the original code here: https://github.com/mikeferguson/simple grasping. This package is quite handy for generating grip positions for numerous item kinds. And its simplicity facilitates the process of learning. This package obtains the graspable object's position from the 3D camera sensor (PointCloud) and generates the grasping sequences required to pick up the object.
Now, within the package, there are two folders: scripts and src. Here is where the package's primary programs are situated. In this chapter, however, we will concentrate on the two most crucial:

- basic_grasping_perception.cpp: As you can see, this script is written in C++. Its primary role is to detect an object to be grasped utilizing data from the 3D camera mounted on the head of the Fetch robot.
- pick_and_place.py: As can be seen, this script is written in Python. Its primary duty is to design the motions necessary to first pick up the identified object on the table and then place it in a different location on the table. Obviously, it will utilize the already-created MoveIt package to plan all motion.

In this section of the paper, we will examine these two scripts and their operation in further detail.

The basic_grasping_perception.cpp script begins by creating a launch file that will initiate the ROS node. Additionally, we load the simple_grasping.yaml file, which will provide specific gripper information.

```
  yaml file for package simplegrasping with rosnode "
    basicgraspingperception"


gripper:
  tooltoplanningframe: 0.148
  fingerdepth: 0.038
  grippertolerance: 0.05
  approach:
    min: 0.145
    desired: 0.15
  retreat:
    min: 0.145
    desired: 0.15

usedebug: True
```

Consequently, this file primarily contains information about the gripper. These parameters can be adjusted to enhance the grasping procedure. The following are the most significant parameters we are defining here:

- **tool_to_planning_frame:** The distance from the tool (gripper) to the planning frame of the arm. Usually it is the last link of the arm, in this case, wrist_roll_link.
- **gripper_tolerance:** The allowed error for the grasping process.
- **approach:** The distance to move the gripper when approaching the object to grasp.
- **retreat:** The distance to move the gripper back once it has grasped the object.
- **use_debug:** This parameter will trigger the debug mode, which will allow us to visualize the perception process through RViz.

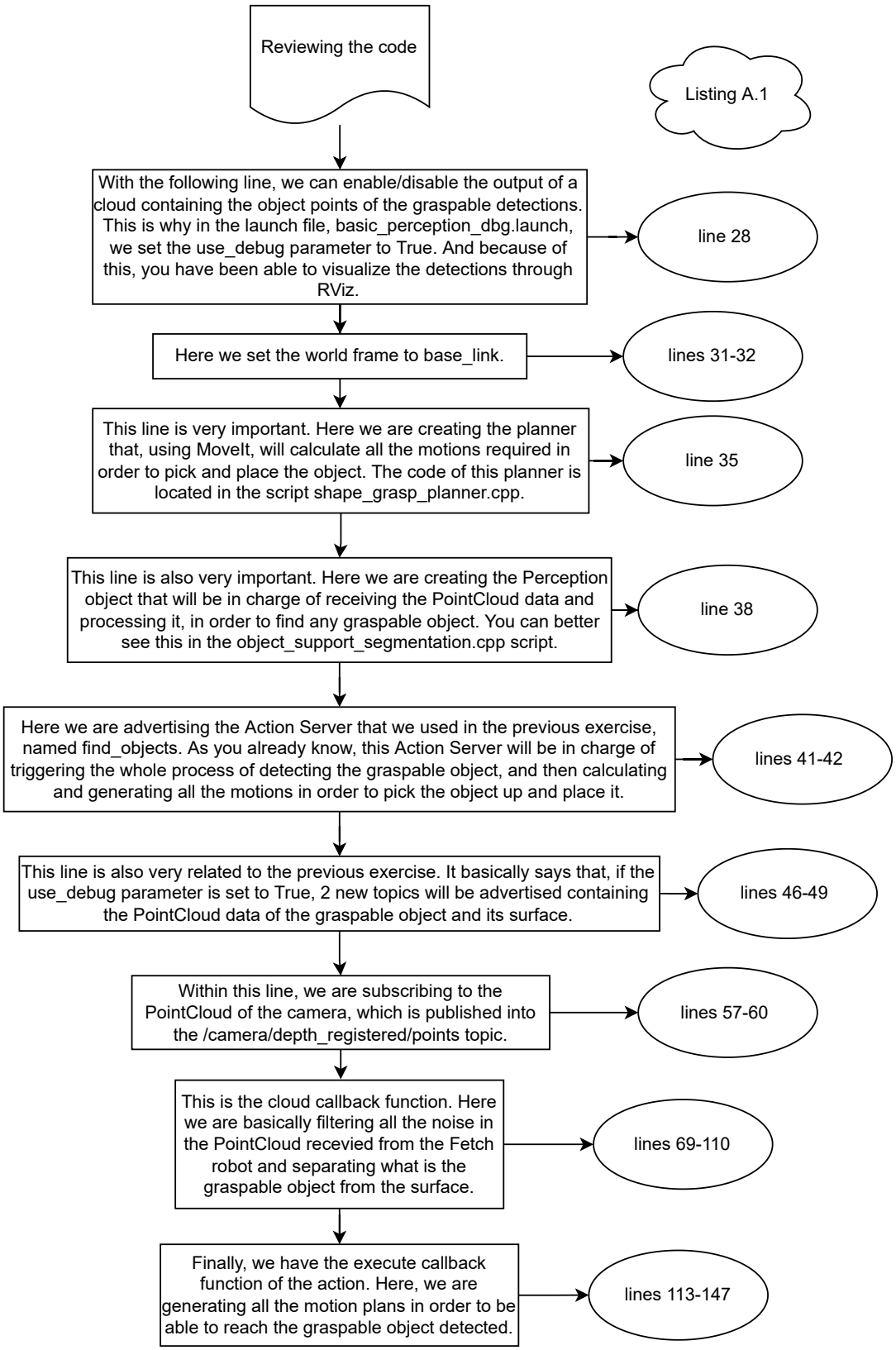

Figure 4.1: Most Relevant Parts of the code

4.1.1 Creating a pick and place task

Pick and place activities can be completed in a variety of ways. For example, we might immediately transmit the robot a specified series of joint values, causing the robot to constantly do the same predefined action (Yang et al., 2020). As a result, we must constantly place the object in the same position in this situation. This approach is known as forward kinematics because it requires prior knowledge of the sequence of joint values necessary to accomplish a certain trajectory.

Another approach would be to use inverse kinematics (IK) in the absence of visual feedback. In this scenario, we supply the robot with the pose (x, y, and z) of the object to be picked up, and the robot knows the motions it has to do to attain the object's posture by performing some IK calculations (Collet et al., 2009).

Finally, another approach would be to apply inverse kinematics with vision assistance or feedback. In this situation, we would utilize a node to determine the posture of the item by reading data from the robot's sensors, such as a Kinect camera. The vision node would then communicate the item's posture, and the robot would know the appropriate motions to do in order to approach the object using IK calculations.

You're probably thinking that the third way is the most efficient and stylish, and you're completely correct. So, for our approach, we will employ the third way. We examined all of the Perception aspects that will allow us to identify the thing we want to grip in the first half of this chapter, so let's now focus on the grasping part!

Inside the simple_grasping package, we also had another important script, which is called pick_and_place.py. After executing the Python code, a number of events will transpire; thus, let's break it down!

Initially, if you opened the MoveIt RViz node, you observed a few green blocks emerge on the RViz display, correct? Do you have any idea what those are?

Obviously, the table and the graspable object were involved! These two items have been generated and put to the MoveIt Planning Scene within the pick and place.py script. Whenever an object is added to the Planning Scene, it will appear in this fashion.

The Pick Action is then initiated. After receiving the grasp object's position, our node transmits it to the grasp server, which generates IK and checks for valid IK to pick up the item. If it discovers a viable IK solution, the arm will initiate the prescribed

motions to pick up the object if it is able to do so.

The Place Action begins once the object has been grasped by the gripper. Again, our node will transmit a Pose to the grab server to indicate where the arm should place the object. The server will then check for a valid IK solution for the posture supplied. If it discovers a viable answer, the gripper will shift to that place and release the object. You can view /grasp and /place to have a better understanding of what is occurring.

4.1.2 rqt Graph

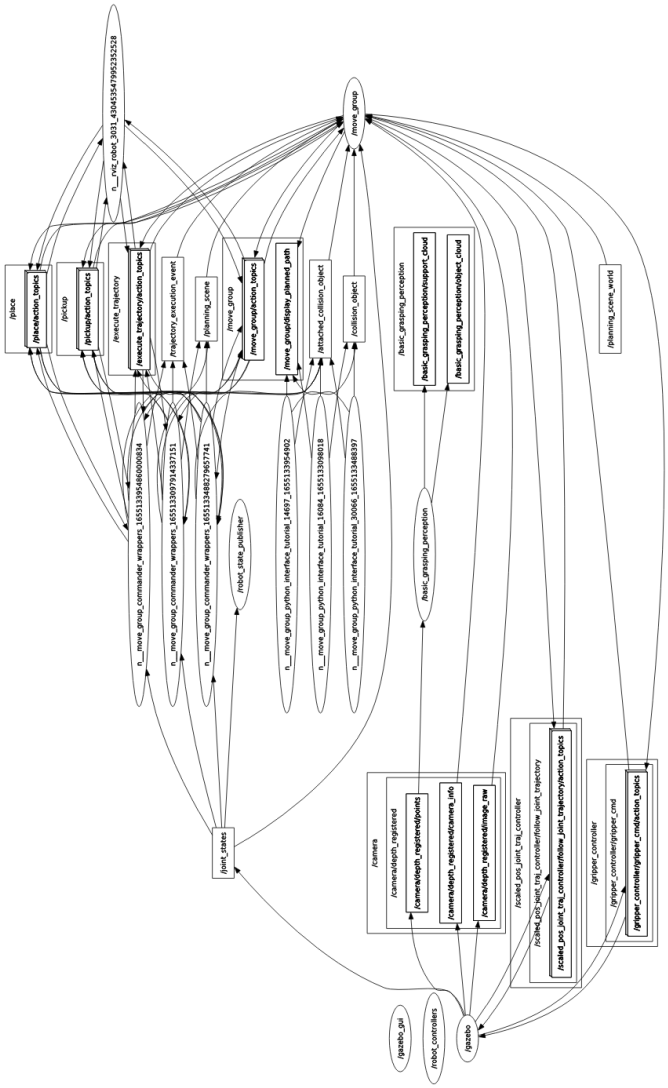

Figure 4.2: All Topics rqt_graph

4.1.3 Simulations

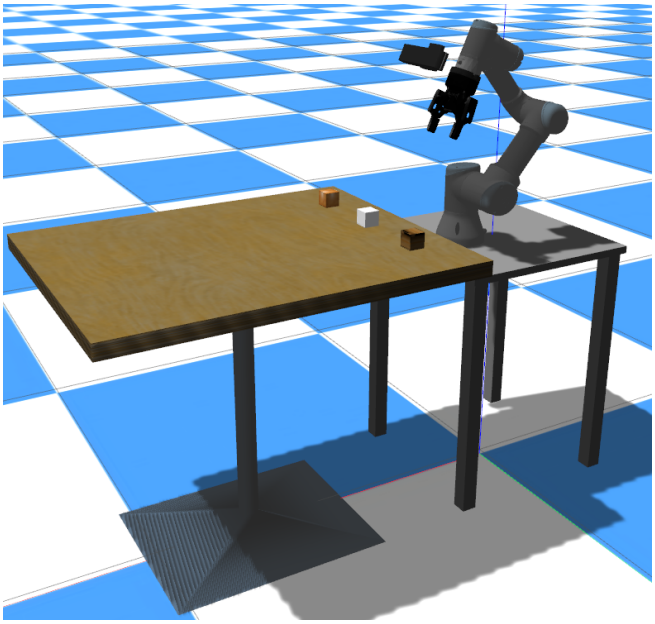

Figure 4.3: Image of the initial pose

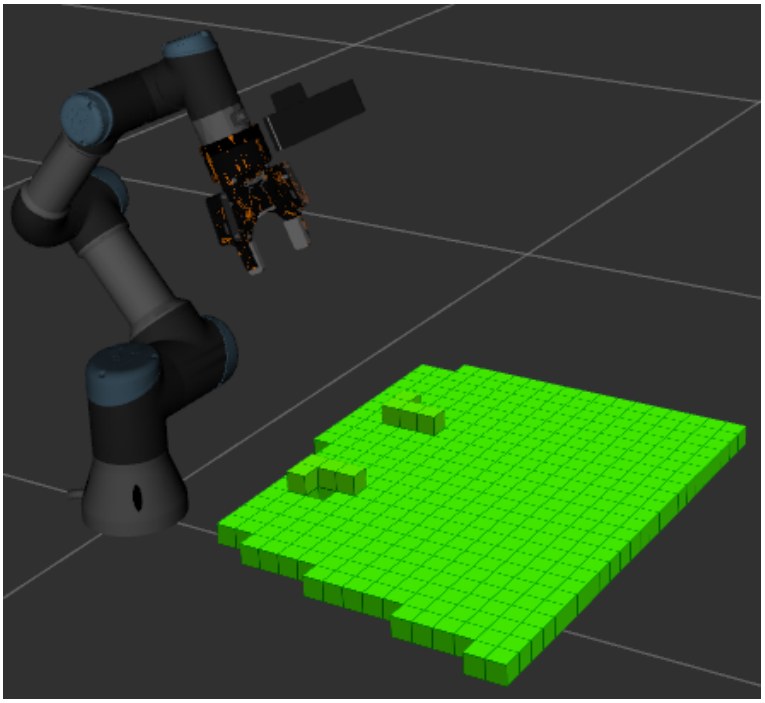

Figure 4.4: Octomap generation by camera topics on rviz

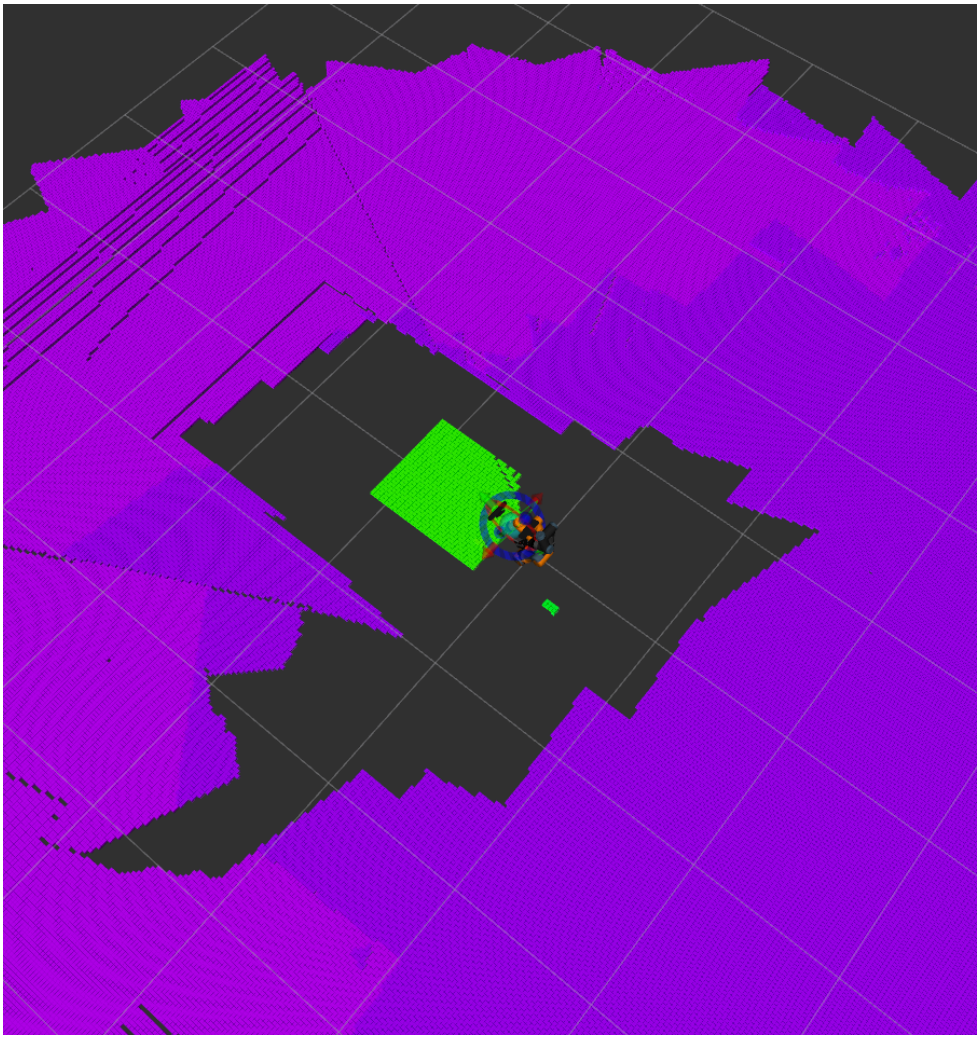

Figure 4.5: Environment Octomap generation by camera topics on rviz

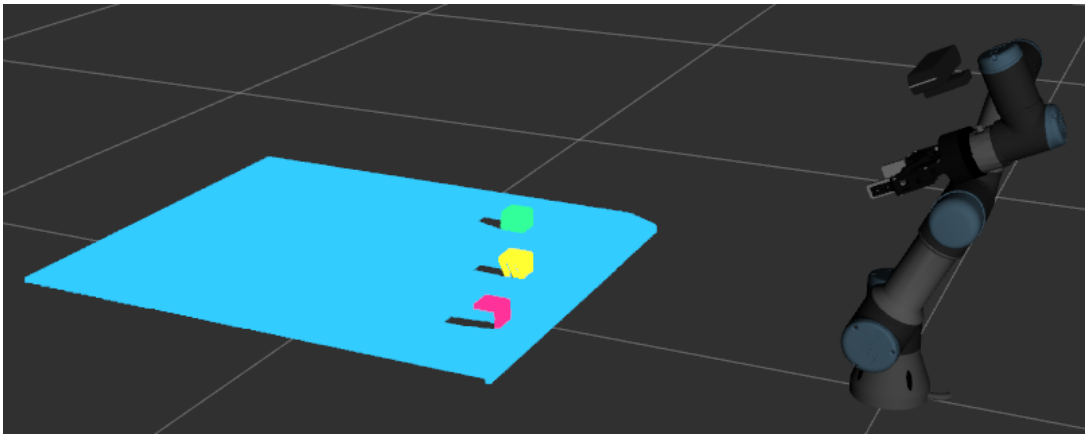

Figure 4.6: Object and Support detection by Perception node

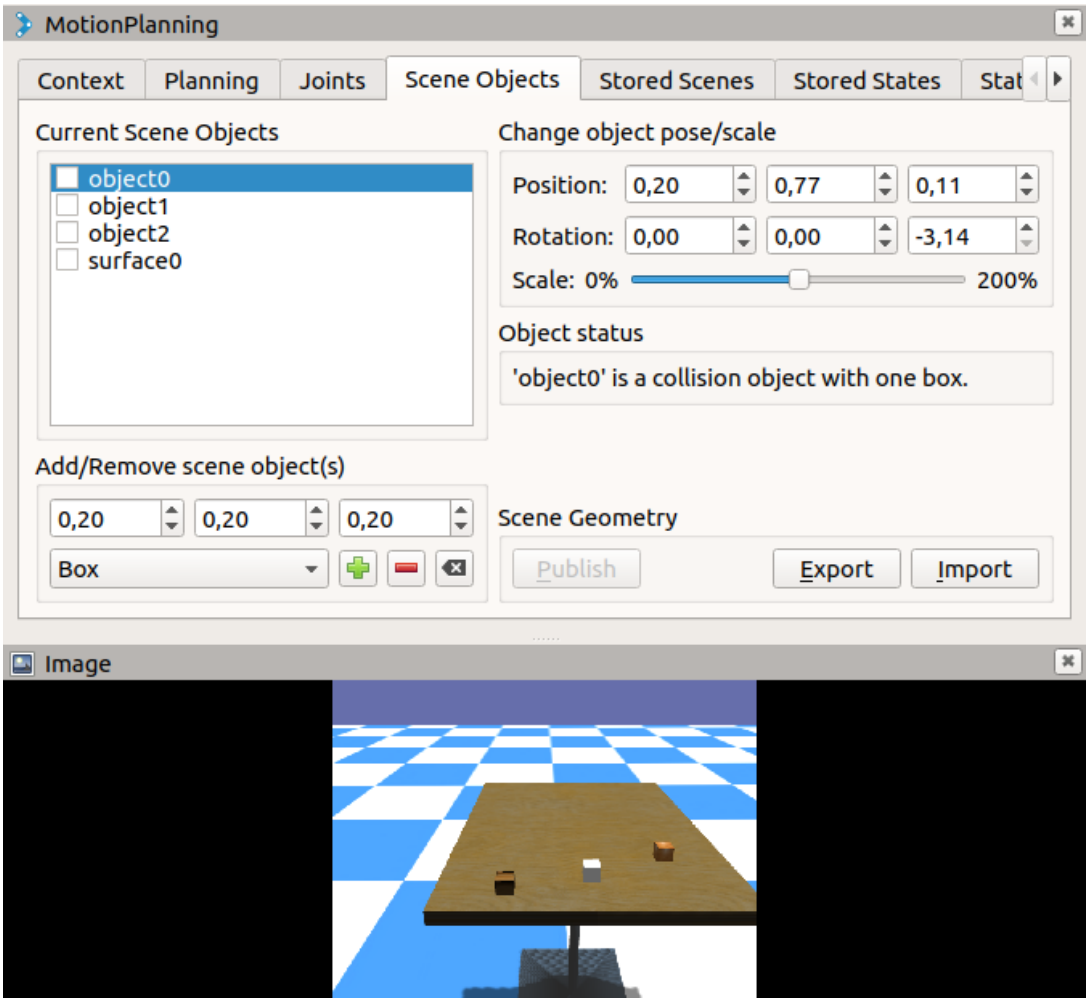

Figure 4.7: Scene Objects on MotionPlanning tool

The computer's hardware, on which the simulation is running, is critical. A repeatable and reliable test environment may not be possible due to the lack of optimization and compute capacity during the creation of algorithms. Simulators that are appropriate for soft robots can be used in settings created for traditional robotics to create a repeatable simulation platform for hybrid robots on more powerful hardware.

CHAPTER 5

CONCLUSION

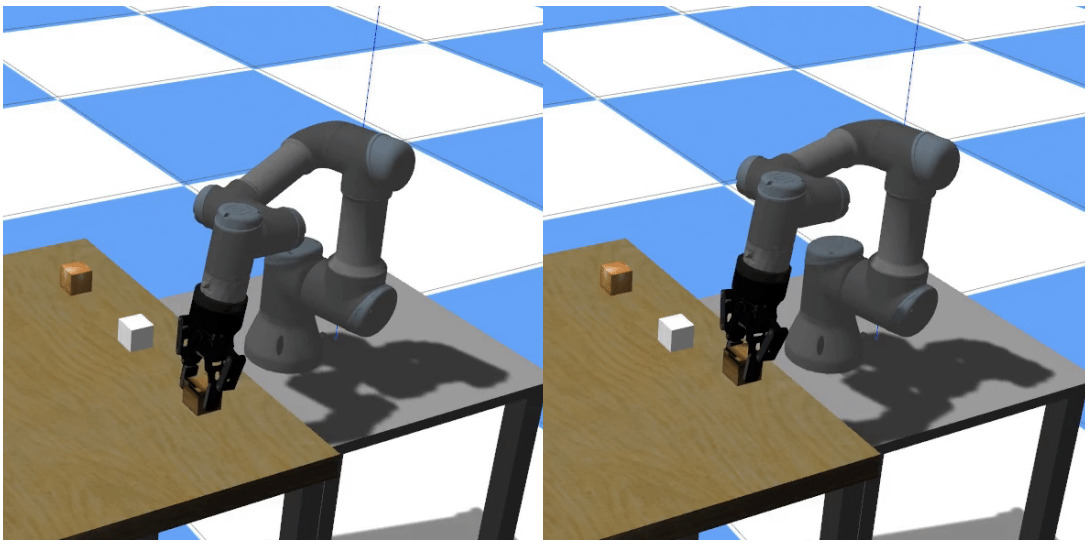

Figure 5.1: Pick and Place

In this paper we analyzed the possible models of a mobile manipulator. However, the focus was more on the robotic arm. The URDF (3D Model on ROS) was constructed and a Moveit! package was built upon it. Which includes the collision matrix, the joints definitions, ROS controllers, 3D perception (sensors definition), and solver selection. This package is then used to assist in motion planning through Rviz. However, for the Pick and Place mission, the motion planning had to be done programmatic-ally using python. Where a C++ code that acts as a perception node was able to detect objects and surfaces that could later be forward to the Pick and Place node for further processing. Note that even though motion planning is done programmatic-ally, the move group node is still active in terms of trajectory planning and poses calculations.

REFERENCES

Aaltonen, M., & Barth, T. (2005). How Do We Make Sense of the Future? *Journal of Futures Studies*, *9*(4), 45–60.

Bogue, R. (2012). Robots in the laboratory: A review of applications. *Industrial Robot*, *39*, 113–119. https://doi.org/10.1108/01439911211203382

Causo, A., Chong, Z. H., Luxman, R., Kok, Y. Y., Yi, Z., Pang, W. C., Meixuan, R., Teoh, Y. S., Jing, W., Tju, H. S., & Chen, I. M. (2018). A robust robot design for item picking. *Proceedings - IEEE International Conference on Robotics and Automation*, 7421–7426. https://doi.org/10.1109/ICRA.2018.8461057

Choi, B. J., You, W. S., Shin, S. H., Moon, H., Koo, J. C., Chung, W., & Choi, H. R. (2011). Development of robotic laboratory automation platform with intelligent mobile agents for clinical chemistry. *IEEE International Conference on Automation Science and Engineering*, 708–713. https://doi.org/10.1109/CASE.2011.6042468

Collet, A., Berenson, D., Srinivasa, S. S., & Ferguson, D. (2009). Object recognition and full pose registration from a single image for robotic manipulation. *Proceedings - IEEE International Conference on Robotics and Automation*, 48–55. https://doi.org/10.1109/ROBOT.2009.5152739

Denavit, J., & Hartenberg, R. S. (1955). A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, *22*. https://doi.org/10.1115/1.4011045

Jara, C. A., Candelas, F. A., & Torres, F. (2008). Virtual and remote laboratory for robotics e-learning. *Computer Aided Chemical Engineering*, *25*, 1193–1198. https://doi.org/10.1016/S1570-7946(08)80205-2

Kumar, V., Wang, Q., Minghua, W., Rizwan, S., Shaikh, S. M., & Liu, X. (2018). Computer vision based object grasping 6dof robotic arm using picamera. *Proceedings - 2018 4th International Conference on Control, Automation and*

Robotics, ICCAR 2018, 111–115. https://doi.org/10.1109/ICCAR.2018.8384653

Mahtani, A., Luis, S., Enrique, F., Aaron, M., & Lentin, J. (2018). *Mastering ros for robotics programming: Design, build, and simulate complex robots using the robot operating system*.

Whitty, M. (2012). Robotics, vision and control. fundamental algorithms in matlab. *Industrial Robot: An International Journal*, *39*. https://doi.org/10.1108/ir.2012.04939faa.005

Wise, M., Ferguson, M., King, D., Diehr, E., & Dymesich, D. (n.d.). *Fetch & freight: Standard platforms for service robot applications*.

Yang, J.-Y., Chen, U.-K., Chang, K.-C., & Chen, Y.-J. (2020). A novel robotic grasp detection technique by integrating yolo and grasp detection deep neural networks. *2020 International Conference on Advanced Robotics and Intelligent Systems (ARIS)*. https://doi.org/10.1109/aris50834.2020.9205791

Appendix A

PERCEPTION NODE

```
1  include  ros/ros.h
2  include  actionlib/server/simpleactionserver.h
3  include  tf/transformlistener.h
4
5  include  simplegrasping/objectsupportsegmentation.h
6  include  simplegrasping/shapegraspplanner.h
7  include  graspingmsgs/FindGraspableObjectsAction.h
8
9  include  pclros/pointcloud.h
10 include  pclros/transforms.h
11 include  pcl/filters/passthrough.h
12 include  pclconversions/pclconversions.h
13
14 namespace simplegrasping
15
16
17 /
18     brief ROS wrapper for shape grasp planner + object support
     segmentation
19  /
20 class BasicGraspingPerception
21
22  typedef actionlib::SimpleActionServer graspingmsgs::
     FindGraspableObjectsAction  servert;
23
24 public:
25  BasicGraspingPerception(ros::NodeHandle n) : nh(n), debug(false),
     findobjects(false)
26
27    // usedebug: enable/disable output of a cloud containing object points
```

```
28    nh.getParam("usedebug", debug);
29
30    // frameid: frame to transform cloud to (should be XY horizontal)
31    worldframe = "baselink";
32    nh.getParam("frameid", worldframe);
33
34    // Create planner
35    planner.reset(new ShapeGraspPlanner(nh));
36
37    // Create perception
38    segmentation.reset(new ObjectSupportSegmentation(nh));
39
40    // Advertise an action for perception + planning
41    server.reset(new servert(nh, "findobjects",
42                             boost::bind(BasicGraspingPerception::
      executeCallback, this, 1),
43                             false));
44
45    // Publish debugging views
46    if (debug)
47
48      objectcloudpub = nh.advertise<pcl::PointCloud<pcl::PointXYZRGB>
      >("objectcloud", 1);
49      supportcloudpub = nh.advertise<pcl::PointCloud<pcl::PointXYZRGB>
      >("supportcloud", 1);
50
51
52    // Range filter for cloud
53    rangefilter.setFilterFieldName("z");
54    rangefilter.setFilterLimits(0, 2.5);
55
56    // Subscribe to camera cloud
57    cloudsub = nh.subscribe<pcl::PointCloud<pcl::PointXYZRGB>>("/
      camera/depthregistered/points",
58                                                               1,
59
      BasicGraspingPerception::cloudCallback,
60                                                               this);
61
62    // Start thread for action
```

```
63      server  start();
64
65      ROSINFO("basicgraspingperception initialized");
66
67
68  private:
69    void cloudCallback(const pcl::PointCloud pcl::PointXYZRGB::ConstPtr
        cloud)
70
71      // be lazy
72      if (!findobjects)
73        return;
74
75      ROSDEBUG("Cloud recieved with  d points.", staticcast int (cloud
        points.size()));
76
77      // Filter out noisy long range points
78      pcl::PointCloud pcl::PointXYZRGB::Ptr cloudfiltered(new pcl::
        PointCloud pcl::PointXYZRGB );
79      rangefilter.setInputCloud(cloud);
80      rangefilter.filter( cloudfiltered);
81      ROSDEBUG("Filtered for range, now  d points.", staticcast int (
        cloudfiltered  points.size()));
82
83      // Transform to grounded
84      pcl::PointCloud pcl::PointXYZRGB::Ptr cloudtransformed(new pcl::
        PointCloud pcl::PointXYZRGB );
85      if (!pclros::transformPointCloud(worldframe,  cloudfiltered,
        cloudtransformed, listener))
86
87        ROSERROR("Error transforming to frame  s", worldframe.cstr());
88        return;
89
90
91      // Run segmentation
92      objects.clear();
93      supports.clear();
94      pcl::PointCloud pcl::PointXYZRGB  objectcloud;
95      pcl::PointCloud pcl::PointXYZRGB  supportcloud;
96      if (debug)
```

```
97
98      objectcloud.header.frameid = cloudtransformed  header.frameid;
99      supportcloud.header.frameid = cloudtransformed  header.frameid;
100
101    segmentation  segment(cloudtransformed, objects, supports,
       objectcloud, supportcloud, debug);
102
103    if (debug)
104
105      objectcloudpub.publish(objectcloud);
106      supportcloudpub.publish(supportcloud);
107
108
109    // Ok to continue processing
110    findobjects = false;
111
112
113  void executeCallback(const graspingmsgs::
       FindGraspableObjectsGoalConstPtr  goal)
114
115    graspingmsgs::FindGraspableObjectsResult result;
116
117    // Get objects
118    findobjects = true;
119    ros::Time t = ros::Time::now();
120    while (findobjects == true)
121
122      ros::Duration(1/50.0).sleep();
123      if (ros::Time::now()   t   ros::Duration(3.0))
124
125        findobjects = false;
126        server  setAborted(result, "Failed to get camera data in alloted
       time.");
127        ROSERROR("Failed to get camera data in alloted time.");
128        return;
129
130
131
132    // Set object results
133    for (sizet i = 0; i   objects.size(); ++i)
```

```
134
135       graspingmsgs::GraspableObject g;
136       g.object = objects[i];
137       if (goal  plangrasps)
138
139         // Plan grasps for object
140         planner  plan(objects[i], g.grasps);
141
142       result.objects.pushback(g);
143
144     // Set support surfaces
145     result.supportsurfaces = supports;
146
147     server  setSucceeded(result, "Succeeded.");
148
149
150   ros::NodeHandle nh;
151
152   bool debug;
153
154   tf::TransformListener listener;
155   std::string worldframe;
156
157   bool findobjects;
158   std::vector graspingmsgs::Object  objects;
159   std::vector graspingmsgs::Object  supports;
160
161   ros::Subscriber cloudsub;
162   ros::Publisher objectcloudpub;
163   ros::Publisher supportcloudpub;
164
165   boost::sharedptr ShapeGraspPlanner  planner;
166   boost::sharedptr ObjectSupportSegmentation  segmentation;
167
168   boost::sharedptr servert  server;
169
170   pcl::PassThrough pcl::PointXYZRGB  rangefilter;
171 ;
172
173    // namespace simplegrasping
```

```
174
175 int main(int argc, char  argv[])
176
177   ros::init(argc, argv, "basicgraspingperception");
178   ros::NodeHandle n(" ");
179   simplegrasping::BasicGraspingPerception perceptionnplanning(n);
180   ros::spin();
181   return 0;
182
```

Listing A.1: basic_grasping_perception.cpp

```
1  include  Eigen/Eigen
2  include  ros/ros.h
3  include  simplegrasping/shapegraspplanner.h
4
5 using shapemsgs::SolidPrimitive;
6
7 namespace simplegrasping
8
9
10 moveitmsgs::GripperTranslation makeGripperTranslation(
11   std::string frame,
12   double min,
13   double desired,
14   double xaxis = 1.0,
15   double yaxis = 0.0,
16   double zaxis = 0.0)
17
18   moveitmsgs::GripperTranslation translation;
19   translation.direction.vector.x = xaxis;
20   translation.direction.vector.y = yaxis;
21   translation.direction.vector.z = zaxis;
22   translation.direction.header.frameid = frame;
23   translation.mindistance = min;
24   translation.desireddistance = desired;
25   return translation;
26
27
28 Eigen::Quaterniond quaternionFromEuler(float yaw, float pitch, float roll)
29
```

```
30   float sy = sin(yaw*0.5);
31   float cy = cos(yaw*0.5);
32   float sp = sin(pitch*0.5);
33   float cp = cos(pitch*0.5);
34   float sr = sin(roll*0.5);
35   float cr = cos(roll*0.5);
36   float w = cr*cp*cy + sr*sp*sy;
37   float x = sr*cp*cy - cr*sp*sy;
38   float y = cr*sp*cy + sr*cp*sy;
39   float z = cr*cp*sy - sr*sp*cy;
40   return Eigen::Quaterniond(w,x,y,z);
41
42
43 ShapeGraspPlanner::ShapeGraspPlanner(ros::NodeHandle& nh)
44
45   /
46     Gripper model is based on having two fingers, and assumes
47     that the robot is using the moveit_simple_controller_manager
48     gripper interface, with "parallel" parameter set to true.
49    /
50   nh.param<std::string>("gripper/left_joint", left_joint_, "
       robotiq_85_left_knuckle_joint");
51   nh.param<std::string>("gripper/right_joint", right_joint_, "
       robotiq_85_right_knuckle_joint");
52   nh.param("gripper/max_opening", max_opening_, 0.095);
53   nh.param("gripper/max_effort", max_effort_, 100.0);
54   nh.param("gripper/finger_depth", finger_depth_, 0.038);
55   nh.param("gripper/grasp_duration", grasp_duration_, 2.0);
56   nh.param("gripper/gripper_tolerance", gripper_tolerance_, 0.05);
57
58   /
59     Approach is usually aligned with wrist_roll
60    /
61   nh.param<std::string>("gripper/approach/frame", approach_frame_, "
       wrist_3_link");
62   nh.param("gripper/approach/min", approach_min_translation_, 0.1);
63   nh.param("gripper/approach/desired", approach_desired_translation_, 0.15)
       ;
64
65   /
```

```
66      Retreat is usually aligned with wristroll
67    /
68   nh.param std::string ("gripper/retreat/frame", retreatframe, "
       wrist3link");
69   nh.param("gripper/retreat/min", retreatmintranslation, 0.1);
70   nh.param("gripper/retreat/desired", retreatdesiredtranslation, 0.15);
71
72   // Distance from tool point to planning frame
73   nh.param("gripper/tooltoplanningframe", tooloffset, 0.148);
74
75
76 int ShapeGraspPlanner::createGrasp(const geometrymsgs::PoseStamped  pose,
77                                    double gripperopening,
78                                    double gripperpitch,
79                                    double xoffset,
80                                    double zoffset,
81                                    double quality)
82
83   moveitmsgs::Grasp grasp;
84   grasp.grasppose = pose;
85
86   // defaults
87   grasp.pregrasppposture = makeGraspPosture(gripperopening);
88   grasp.grasppposture = makeGraspPosture(0.0);
89   grasp.pregraspapproach = makeGripperTranslation(approachframe,
90
       approachmintranslation,
91
       approachdesiredtranslation);
92   grasp.postgraspretreat = makeGripperTranslation(retreatframe,
93
       retreatmintranslation,
94
       retreatdesiredtranslation,
95                                                   1.0);  // retreat is
       in negative x direction
96
97   // initial pose
98   Eigen::Affine3d p = Eigen::Translation3d(pose.pose.position.x,
99                                            pose.pose.position.y,
```

```
                                          pose.pose.position.z)
                        Eigen::Quaterniond(pose.pose.orientation.w,
                                          pose.pose.orientation.x,
                                          pose.pose.orientation.y,
                                          pose.pose.orientation.z);
  // translate by xoffset, 0, zoffset
  p = p   Eigen::Translation3d(xoffset, 0, zoffset);
  // rotate by 0, pitch, 0
  p = p   quaternionFromEuler(0.0, gripperpitch, 0.0);
  // apply grasp point    planning frame offset
  p = p   Eigen::Translation3d( tooloffset , 0, 0);

  grasp.grasppose.pose.position.x = p.translation().x();
  grasp.grasppose.pose.position.y = p.translation().y();
  grasp.grasppose.pose.position.z = p.translation().z();
  Eigen::Quaterniond q = (Eigen::Quaterniond)p.linear();
  grasp.grasppose.pose.orientation.x = q.x();
  grasp.grasppose.pose.orientation.y = q.y();
  grasp.grasppose.pose.orientation.z = q.z();
  grasp.grasppose.pose.orientation.w = q.w();

  grasp.graspquality = quality;

  grasps.pushback(grasp);
  return 1;


// Create the grasps going in one direction around an object
// starts with gripper level, rotates it up
// this works for boxes and cylinders
int ShapeGraspPlanner::createGraspSeries(
  const geometrymsgs::PoseStamped  pose,
  double depth, double width, double height,
  bool usevertical)

  int count = 0;

  // Gripper opening is limited
  if (width  = (maxopening 0.9))
    return count;
```

```
140
141 // Depth of grasp calculations
142 double x = depth/2.0;
143 double z = height/2.0;
144 if (x   fingerdepth)
145   x = fingerdepth   x;
146 if (z   fingerdepth)
147   z = fingerdepth   z;
148
149 double open = std::min(width + grippertolerance, maxopening);
150
151 // Grasp along top of box
152 for (double step = 0.0; step   depth/2.0; step += 0.1)
153
154   if (usevertical)
155     count += createGrasp(pose, open, 1.57, step,  z, 1.0   0.1 step);  //
     vertical
156   count += createGrasp(pose, open, 1.07, step,  z + 0.01, 0.7   0.1 step)
    ;  // slightly angled down
157   if (step   0.05)
158
159     if (usevertical)
160       count += createGrasp(pose, open, 1.57,  step,  z, 1.0   0.1 step);
161     count += createGrasp(pose, open, 1.07,  step,  z + 0.01, 0.7   0.1
    step);
162
163
164
165 // Grasp horizontally along side of box
166 for (double step = 0.0; step   height/2.0; step += 0.1)
167
168   count += createGrasp(pose, open, 0.0, x, step, 0.8   0.1 step);  //
    horizontal
169   count += createGrasp(pose, open, 0.5, x0.01, step, 0.6   0.1 step);
    // slightly angled up
170   if (step   0.05)
171
172     count += createGrasp(pose, open, 0.0, x,  step, 0.8   0.1 step);
173     count += createGrasp(pose, open, 0.5, x0.01,  step, 0.6   0.1 step);
174
```

```
175
176
177   // A grasp on the corner of the box
178   count += createGrasp(pose, open, 1.57/2.0, x   0.005,  z + 0.005, 0.25);
179
180   return count;
181
182
183 int ShapeGraspPlanner::plan(const graspingmsgs::Object  object,
184                             std::vector moveitmsgs::Grasp   grasps)
185
186   ROSINFO("shape grasp planning starting...");
187
188   // Need a shape primitive
189   if (object.primitives.size() == 0)
190
191     // Shape grasp planner can only plan for objects
192     //  with SolidPrimitive bounding boxes
193     return  1;
194
195
196   if (object.primitiveposes.size() != object.primitives.size())
197
198     // Invalid object
199     return  1;
200
201
202   // Clear out internal vector
203   grasps.clear();
204
205   // Setup Pose
206   geometrymsgs::PoseStamped grasppose;
207   grasppose.header = object.header;
208   grasppose.pose = object.primitiveposes[0];
209
210   // Setup object orientation
211   Eigen::Quaterniond q(object.primitiveposes[0].orientation.w,
212                        object.primitiveposes[0].orientation.x,
213                        object.primitiveposes[0].orientation.y,
214                        object.primitiveposes[0].orientation.z);
```

```
215
216 // Setup object dimensions
217 double x, y, z;
218 if (object.primitives[0].type == SolidPrimitive::BOX)
219
220   x = object.primitives[0].dimensions[SolidPrimitive::BOXX];
221   y = object.primitives[0].dimensions[SolidPrimitive::BOXY];
222   z = object.primitives[0].dimensions[SolidPrimitive::BOXZ];
223
224 else if (object.primitives[0].type == SolidPrimitive::CYLINDER)
225
226   x = y = 2.0   object.primitives[0].dimensions[SolidPrimitive::
     CYLINDERRADIUS];
227   z = object.primitives[0].dimensions[SolidPrimitive::CYLINDERHEIGHT];
228
229
230 // Generate grasps
231 for (int i = 0; i   4; ++i)
232
233   grasppose.pose.orientation.x = q.x();
234   grasppose.pose.orientation.y = q.y();
235   grasppose.pose.orientation.z = q.z();
236   grasppose.pose.orientation.w = q.w();
237
238   if (i   2)
239
240     createGraspSeries(grasppose, x, y, z);
241
242   else
243
244     // Only two sets of unqiue vertical grasps
245     createGraspSeries(grasppose, x, y, z, false);
246
247
248   // Next iteration, rotate 90 degrees about Z axis
249   q = q   quaternionFromEuler(1.57, 0.0, 0.0);
250   std::swap(x, y);
251
252
253 ROSINFO("shape grasp planning done.");
```

```
254
255   grasps = grasps;
256   return grasps.size();  // num of grasps
257
258
259 trajectorymsgs::JointTrajectory
260 ShapeGraspPlanner::makeGraspPosture(double pose)
261
262   trajectorymsgs::JointTrajectory trajectory;
263   trajectory.jointnames.pushback(leftjoint);
264   trajectory.jointnames.pushback(rightjoint);
265   trajectorymsgs::JointTrajectoryPoint point;
266   point.positions.pushback(pose/2.0);
267   point.positions.pushback(pose/2.0);
268   point.effort.pushback(maxeffort);
269   point.effort.pushback(maxeffort);
270   point.timefromstart = ros::Duration(graspduration);
271   trajectory.points.pushback(point);
272   return trajectory;
273
274
275    // namespace simplegrasping
```

Listing A.2: shape_grasp_planner.cpp

```
1  !/usr/bin/env python
2
3  import argparse
4  import copy
5  import math
6  import sys
7
8  import rospy
9  import actionlib
10 from moveitpython import
11 from moveitpython.geometry import rotateposemsgbyeulerangles
12
13 from graspingmsgs.msg import
14 from moveitmsgs.msg import MoveItErrorCodes, PlaceLocation
15
16 jointnames = ["shoulderpanjoint", "shoulderliftjoint", "elbowjoint",
```

```
    "wrist1joint", "wrist2joint", "wrist3joint"]
readypose = [1.57, 2.27, 1.40, 1.13, 1.57, 0]

def movetoready(interface):
   result = interface.moveToJointPosition(jointnames, readypose)
   if result.errorcode.val != 1:
       rospy.sleep(1.0)
       rospy.logerr("Move arm to ready position failed, trying again...")
       result = interface.moveToJointPosition(jointnames, readypose,
    0.02)

if  name ==" main ":
    parser = argparse.ArgumentParser(description="Simple demo of pick and
    place")
    parser.addargument(" objects", help="Just do object perception",
    action="storetrue")
    parser.addargument(" all", help="Just do object perception, but
    insert all objects", action="storetrue")
    parser.addargument(" once", help="Run once.", action="storetrue")
    parser.addargument(" ready", help="Move the arm to the ready position
    .", action="storetrue")
    parser.addargument(" plan", help="Only do planning, no execution",
    action="storetrue")
    parser.addargument(" x", help="Recommended x offset, how far out an
    object should roughly be.", type=float, default=0.5)
    args, unknown = parser.parseknownargs()

    rospy.initnode("pickandplacedemo")
    movegroup = MoveGroupInterface("arm", "baselink", planonly = args.
    plan)

      if all we want to do is prepare the arm, do it now
    if args.ready:
        movetoready(movegroup)
        exit(0)

    scene = PlanningSceneInterface("baselink")
    pickplace = PickPlaceInterface("arm", "gripper", planonly = args.plan,
     verbose = True)

```

```
    rospy.loginfo("Connecting to basicgraspingperception/findobjects
    ...")
    findobjects = actionlib.SimpleActionClient("basicgraspingperception/
    findobjects", FindGraspableObjectsAction)
    findobjects.waitforserver()
    rospy.loginfo("...connected")

    while not rospy.isshutdown():
        goal = FindGraspableObjectsGoal()
        goal.plangrasps = True
        findobjects.sendgoal(goal)
        findobjects.waitforresult(rospy.Duration(5.0))
        findresult = findobjects.getresult()

        rospy.loginfo("Found  d objects"   len(findresult.objects))

          remove all previous objects

        for name in scene.getKnownCollisionObjects():
            scene.removeCollisionObject(name, False)
        for name in scene.getKnownAttachedObjects():
            scene.removeAttachedObject(name, False)
        scene.waitForSync()

          clear colors
        scene.colors = dict()

          insert objects, find the one to grasp
        theobject = None
        theobjectdist = 1
        count =  1
        for obj in findresult.objects:
            count += 1
            scene.addSolidPrimitive("object d" count,
                                    obj.object.primitives[0],
                                    obj.object.primitiveposes[0],
                                    useservice = False)
              object must have usable grasps
            if len(obj.grasps)   1:
                continue
```

```
            choose object in front of robot
            dx = obj.object.primitiveposes[0].position.x   args.x
            dy = obj.object.primitiveposes[0].position.y
            d = math.sqrt((dx   dx) + (dy   dy))
            if d   theobjectdist:
                theobjectdist = d
                theobject = count

        if theobject == None:
            rospy.logerr("Nothing to grasp! try again...")
            continue

          insert table
        for obj in findresult.supportsurfaces:
              extend surface to floor
            height = obj.primitiveposes[0].position.z
            obj.primitives[0].dimensions = [obj.primitives[0].dimensions
    [0],
                                            2.0,   make table wider
                                            obj.primitives[0].dimensions[2]
     + height]
            obj.primitiveposes[0].position.z +=  height/2.0

              add to scene
            scene.addSolidPrimitive(obj.name,
                                    obj.primitives[0],
                                    obj.primitiveposes[0],
                                    useservice = False)

        objname = "object d" theobject

          sync
        scene.waitForSync()

          set color of object we are grabbing
        scene.setColor(objname, 223.0/256.0, 90.0/256.0, 12.0/256.0)
    orange
        scene.setColor(findresult.objects[theobject].object.
    supportsurface, 0, 0, 0)    black
        scene.sendColors()
```

```
122
123         exit now if we are just doing object update
124     if args.objects:
125         if args.once:
126             exit(0)
127         else:
128             continue
129
130         get grasps (we checked that they exist above)
131     grasps = findresult.objects[theobject].grasps
132     supportsurface = findresult.objects[theobject].object.
    supportsurface
133
134         call movegroup to pick the object
135     rospy.loginfo("Beginning to pick.")
136     success, pickresult = pickplace.pickwithretry(objname, grasps,
    supportname=supportsurface, scene=scene)
137     if not success:
138         exit(1)
139
140         create a set of place locations for the cube
141     places = list()
142     l = PlaceLocation()
143     l.placepose.pose = findresult.objects[theobject].object.
    primitiveposes[0]
144     l.placepose.header.frameid = findresult.objects[theobject].
    object.header.frameid
145         invert the y of the pose
146     l.placepose.pose.position.y =  l.placepose.pose.position.y
147         copy the posture, approach and retreat from the grasp used
148     l.postplaceposture = pickresult.grasp.pregraspposture
149     l.preplaceapproach = pickresult.grasp.pregraspapproach
150     l.postplaceretreat = pickresult.grasp.postgraspretreat
151     places.append(copy.deepcopy(l))
152         create another several places, rotate each by 90 degrees in yaw
    direction
153     l.placepose.pose = rotateposemsgbyeulerangles(l.placepose.
    pose, 0, 0, 1.57)
154     places.append(copy.deepcopy(l))
155     l.placepose.pose = rotateposemsgbyeulerangles(l.placepose.
```

```
    pose, 0, 0, 1.57)
156         places.append(copy.deepcopy(l))
157         l.placepose.pose = rotateposemsgbyeulerangles(l.placepose.
    pose, 0, 0, 1.57)
158         places.append(copy.deepcopy(l))
159 
160           drop it like it s hot
161         rospy.loginfo("Beginning to place.")
162         while not rospy.isshutdown():
163               can t fail here or moveit needs to be restarted
164             success, placeresult = pickplace.placewithretry(objname,
    places, supportname=supportsurface, scene=scene)
165             if success:
166                 break
167 
168           place arm back at side
169         movetoready(movegroup)
170         rospy.loginfo("Ready...")
171 
172           rinse and repeat
173         if args.once:
174             exit(0)
```

Listing A.3: pick_and_place.py