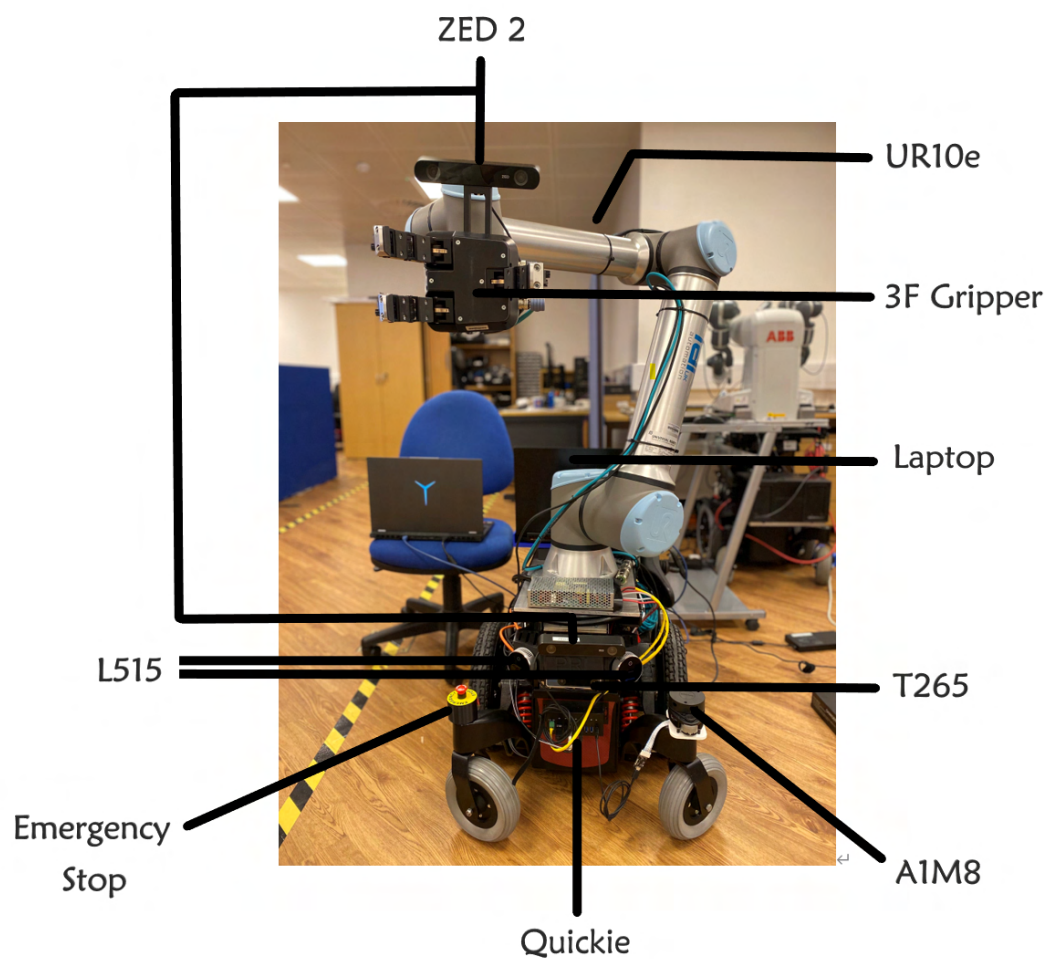


Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2022



Project Title: **Manipulation Skills For A Robot Cooking Assistant**

Student: **Yilong Wang**

CID: **01362420**

Course: **EIE4**

Project Supervisor: **Prof Yiannis Demiris**

Second Marker: **Dr Ad Spiers**

Declaration of Authorship

I declare that this report and the work presented in it are my own. Following the Final Report Plagiarism Statement:

- I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.
- I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.
- I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

Abstract

A robot cooking assistant is among the vast possibilities brought about by the revolutionary development of intelligent robotic automation.

Amiga is a custom-assembled mobile manipulation robot at the personal robotic lab, featuring a UR10e arm, a Robotiq 3F gripper, a mobile Quickie wheelchair base and multiple associated computing devices.

This project aims to develop skills for Amiga so that it may perform tasks in the kitchen just as a cooking assistant would. However, rather than programming analytically for specific tasks, at the end of the project, we have proposed and achieved an engineering architecture that incorporates simulation, perception, manipulation, grasping and navigation such that it can be rapidly built up and extended to carry out new tasks.

Acknowledgements

I would like to take this opportunity to thank:

- My Supervisor Prof. Demiris Yiannis and Senior Tutor Mrs. Perea Esther for their continuous support throughout my fourth year of study at the department. I could not have completed the degree without them.
- All Personal Robotics Lab members, especially Goubard Cédric, who was incredibly patient and helpful to me throughout the project.
- My family, who sponsored my studies at Imperial College and were always supportive of me no matter what decisions I made in life.
- My outstanding friends, who have been an integral aspect of my life and have made it a happy and fulfilling experience.

Amongst all, I would like to thank Tracy, Racheal, Jack, my sister and her friend, who remained by my side during some of my very difficult times at university and have encouraged me more than anything else in this world.

Contents

Declaration of Authorship	i
Abstract	ii
Acknowledgements	iii
1 Introduction	1
1.1 Motivation and Related Work	1
1.2 Objectives and Challenges	3
1.3 Contributions	5
1.4 Outline of Contents	6
2 Preliminary	7
2.1 URDF	7
2.2 Rotation and Transformation	7
2.3 Camera Model	10
2.4 Depth Map and Point Cloud	12
2.5 RANSAC	13
2.6 Deep Learning and CNNs	15
2.7 Wheel Odometry	16
3 Introduction to Amiga	17
3.1 Amiga and Hardware	17
3.1.1 Cameras and Laser Scanners	18
3.1.2 Robotic Arm and Gripper	20
3.1.3 Wheelchair Base and R-NET	21
3.1.4 Embedded Edge Computers	22
3.1.5 Controllers and Networking	23
3.2 ROS and Middleware	24
4 Analysis and System Implementation	26
4.1 Requirement Capture and Analysis	26
4.2 System Overview	27
4.3 Data Workflow	30
5 Amiga Simulation	32
5.1 Background and Module Design	32
5.1.1 Comparing Simulators	32
5.1.2 Simulating Manipulation	33
5.2 Implementation	34

5.2.1	Amiga Robot Model	34
5.2.2	Simulation in Gazebo	36
5.2.3	Camera Plugin	39
5.2.4	MoveIt Integration	40
5.2.5	Grasp Plugin	41
5.2.6	LiDAR plugin	42
5.3	Architecture Summary and Comments	42
6	Amiga Perception	45
6.1	Background and Module Design	45
6.1.1	Object Detection	45
6.1.2	Pose Estimation	47
6.2	Implementation	49
6.2.1	Camera Configuration	49
6.2.2	3D Coordinate Mapping	50
6.2.3	Kitchen Object Detection	50
6.2.4	Secondary Image Processing	54
6.2.5	Object Point Cloud	56
6.2.6	Plane Detection	57
6.2.7	Region Detection	59
6.2.8	Dimension Estimation	61
6.3	Architecture Summary and Comments	64
7	Amiga Manipulation	67
7.1	Background and Module Design	67
7.1.1	Configuration, FK and IK	67
7.1.2	Hand-Eye Calibration	69
7.1.3	Collision-aware Path Planning	71
7.1.4	MoveIt Planning Framework	73
7.2	Implementation	74
7.2.1	Arm Configuration	74
7.2.2	Manipulation Interface	75
7.2.3	Manipulation Server	77
7.2.4	Building Blocks	77
7.2.5	Virtual-effect Translator	79
7.2.6	Grasp Executor	81
7.2.7	Viewpoint Adjuster	82
7.2.8	Vision Participation	84
7.2.9	Obstacle Avoidance	86
7.2.10	Task-study : Stirring	89
7.2.11	Task-study : Pouring	91
7.2.12	Generic Task-solving	92
7.3	Architecture Summary and Comments	93
8	Amiga Grasping	95
8.1	Background and Module Design	95
8.1.1	Grasping Pipeline	95
8.1.2	Planar Grasps and Dexnet	97

8.1.3	6-DoF Grasps and GPD	99
8.2	Implementation	101
8.2.1	Gripper Configuration	101
8.2.2	MoveIt Grasps	102
8.2.3	3-DOF Grasping	103
8.2.4	Replicating GPD	105
8.2.5	Replicating Dex-Net	107
8.2.6	6-DOF Grasping	110
8.2.7	5-DoF Grasping	113
8.2.8	4-DoF Grasping	115
8.2.9	Object-aware Deep Grasp	115
8.2.10	Towards Dexterous Grasping	116
8.2.11	Grasp-strategy Inference	117
8.3	Architecture Summary and Comments	118
9	Amiga Navigation	121
9.1	Background and Module Design	121
9.1.1	Odometry and Localisation	121
9.1.2	SLAM and World Modelling	122
9.1.3	ROS Navigation Stack	123
9.2	Implementation	124
9.2.1	Configuration and Investigation	124
9.2.2	Gmapping and Amcl	126
9.2.3	RTAB-Map and Multi-Sensor Inputs	128
9.3	Architecture Summary and Comments	130
10	Testing and Evaluation	133
10.1	Plane detection	133
10.2	Sim2real	133
10.3	KitchenNet	135
10.4	Motion planning and Grasping	135
11	Conclusions and Further Work	138
11.1	Limitations	138
11.2	Future Work	139
12	User Guide	140
A	Appendix	143
A.1	Amiga Simulation	143
A.1.1	Teleop and Gazebo Worlds	143
A.1.2	Intrinsic and Hand-Eye calibration in Gazebo	145
A.1.3	Arm and Gripper Controllers	147
A.2	Amiga Perception	147
A.2.1	Camera Topics	147
A.2.2	3D Coordinate Mapping	147
A.2.3	Object Detection	148
A.2.4	Foreground Detection GMM	150

A.3	Amiga Manipulation	151
A.3.1	Frame Transforms:	151
A.3.2	Pre-defined Poses	153
	Bibliography	154

1 Introduction

In the past decades, the development of robots has been thriving and successfully deployed products in various industries, including manufacturing, healthcare, agriculture, and house-holding, among others. A robot is a physically-embodied, artificially intelligent device with sensing and actuation [Davison, 2022]. Significant variations exist in the morphology of robots depending on different usage and actuation, which may involve robot arms [SWIFTI, 2020], humanoids [Pepper, 2014], mobile robots [MiR1350, 2021], legged robots [Spot, 2016] and even aerial robots. These miscellaneous types of robots can be combined to create hybrid solutions capable of more sophisticated tasks. The most popular combination as such is a mobile manipulator [XL-GEN, 2016] built from a robot arm manipulator mounted on a mobile platform. As a direct result, such an integrated mobile manipulation system adopts both dexterity and mobility; and its mobile base, in addition, provides an excellent complement for the manipulator by introducing an extended, unlimited portable workspace. Nevertheless, the massive degrees of freedom within the system can render its operation a particular challenge, especially in an unstructured, dynamic environment.

In order to achieve a certain degree of autonomy, the robot must be equipped with relevant skills in acquiring scene knowledge and inferring task-specific decisions. An autonomous mobile manipulation robot (AMMR), as the name suggests, is a mobile manipulator furnished with such prowess. AMMR is an intricate field – the added pose uncertainty introduced by mobility imposes additional constraints on the manipulating problem while limiting the available perception systems and introducing additional constraints on the navigation problem that must now factor in the arm attachment. Albeit challenging, efficient control of AMMRs is acutely rewarding as it allows robots to perform very complex tasks combining perception, manipulation, grasping and navigation.

1.1 Motivation and Related Work

An overview of the development of mobile manipulation robots is presented in Figure 1.1. These systems are designed for applications such as part supply and transport in manufacturing and logistics, object transport and human-robot interaction in personal care. In recent years, there has been a growing demand for mobile manipulator platforms in industrial environments. Despite the fact that fixed-base manipulators might be capable of solving multiple tasks in industrial scenarios, a single mobile platform could handle

multiple tasks in different locations, which could improve the return on investment of a robot, especially for small and medium enterprises that cannot afford multiple static robotic platforms [Roa et al., 2021].

Besides industrial robots, one of the popular applications of mobile manipulators is domestic service robots. PR1 is regarded as a classic example of such robots. With an elementary hand, it can do amazingly useful things such as sweeping, fetching a beer and unloading a dishwasher [Salisbury, 2010]. However, this system is not automated – behind the curtain, a human must control the robots manually to achieve each task. This result is not surprising since, back then, autonomous robotics manipulation was chiefly subjugated to limitations in computer vision and scene comprehension.

Two years after PR1, a fast implementation of deep convolutional neuron networks achieved a superhuman benchmark in a visual pattern recognition contest for the first time [Krizhevsky, Sutskever, and Hinton, 2012], and revolutionized the field of computer vision.

Over the past few years, deep learning has had an undisputed impact on robotic perception, revealing some of the most vexing challenges in manipulating objects outside a laboratory or factory environment. This inevitably leads to a challenging yet exciting problem: can we build a mobile manipulation robot system to assist and interact with humans autonomously?

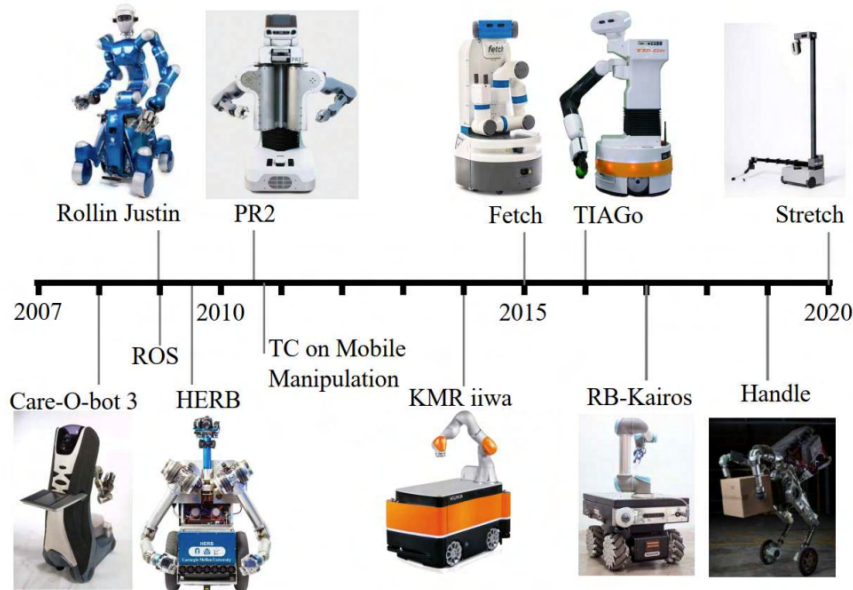


FIGURE 1.1: Timeline for development of mobile robotic manipulators in the last decade [Roa et al., 2021]

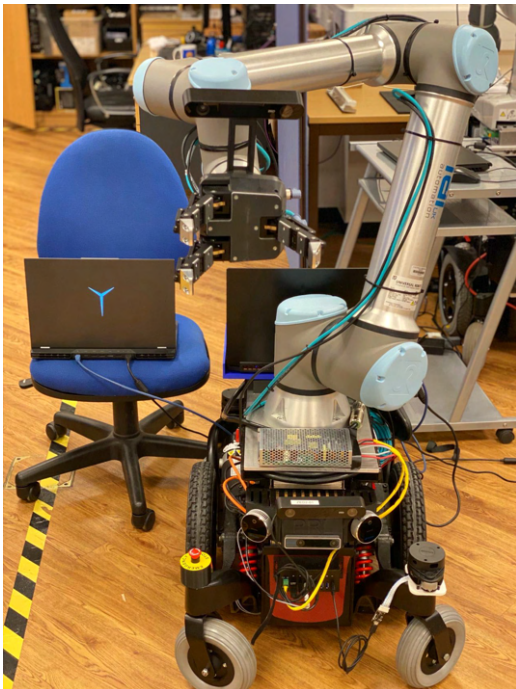
The answer to the last question is undoubtedly affirmative. As a matter of fact, in recent years, autonomous mobile manipulation robots have demonstrated their capabilities at various robot competitions, such as the IROS Mobile Manipulation Hackathon and the International RoboCup, by restocking misplaced goods and carrying out waste in a warehouse [Hackathon, 2018] [RoboCup, 2019]. For the sake of this project, we would like a mobile manipulator robot to complete similar tasks in a kitchen environment.

Cooking is an indispensable aspect of our daily lives. Over time, plenty of companies have developed robotic products in a food service context. Examples include mobile robots like "Servi" from [BearRobotics, 2022], which carries plates from the kitchen to the dining room; manipulation robots such as "Makr Shkr" from [MAKRSHAKR, 2022], which serve as bar bartenders, and "Flippy" from [MisoRobotics, 2022] that can flip hamburgers. More remarkable are the Moley cooking robots [Moley, 2022] consisting of two robot arms that combine and stir-fry pre-prepared ingredients according to recipe instructions. Even so, it is still rare to observe instances of mobile manipulation robots in such a context, hence motivating our work.

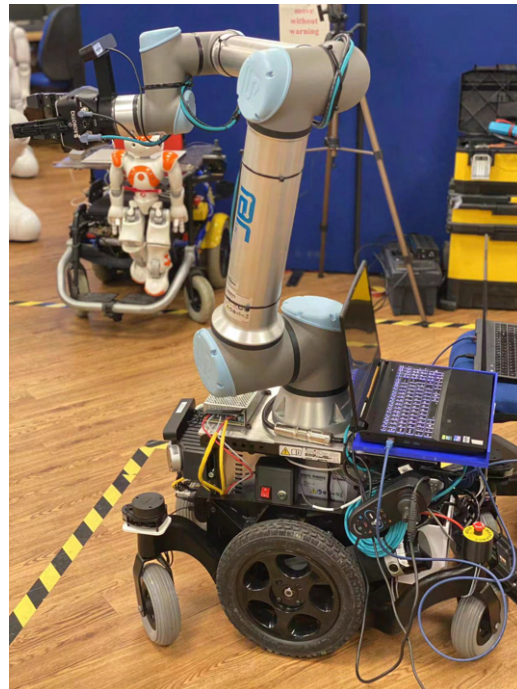
1.2 Objectives and Challenges

As shown in Figure 1.2, Amiga is a custom-assembled mobile manipulation robot at the personal robotics lab, Imperial College London. It features a UR10e arm, a Robotiq 3F gripper, a mobile Quickie wheelchair base and multiple associated computing devices.

This project aims to develop skills for Amiga such that it may perform tasks in a kitchen just as a cooking assistant would. Nevertheless, since the exact tasks sought by the user are not provided, and as we reckon the robot's use would be limited if it could only perform a subset of tasks – instead of programming the robot explicitly for each specific task, we seek to devise a generalised system that will enable developers to adapt to new tasks swiftly.



(A) Amiga front view with many components such as the arm, the gripper, the base and several cameras and LiDAR sensors



(B) Amiga side view with driving wheels, laptops, emergency stops, switches and multiple controlling devices within the base

FIGURE 1.2: Many actuators, sensors and processors in Amiga

Suppose a scene of Amiga under a robot cooking assistant context as found in Figure 1.3. As a part of our efforts to accomplish the objectives, we determined five key sub-modules essential to the success of our system:

1. *Simulation Module* – Amiga should be associated with a simulation environment that allows modules to be tested offline and debugged quickly.
2. *Perception Module* – Amiga should possess a certain degree of knowledge in understanding the semantics of the kitchen environment.
3. *Manipulation Module* – Amiga should plan and execute arm movements in accordance with dynamic obstacles introduced by the mobile base.
4. *Grasping Module* – Amiga should possess a certain degree of knowledge and capabilities in grasping a variety of common kitchen objects.
5. *Navigation Module* – Amiga should be able to localise and navigate around the kitchen without hitting any obstructions.

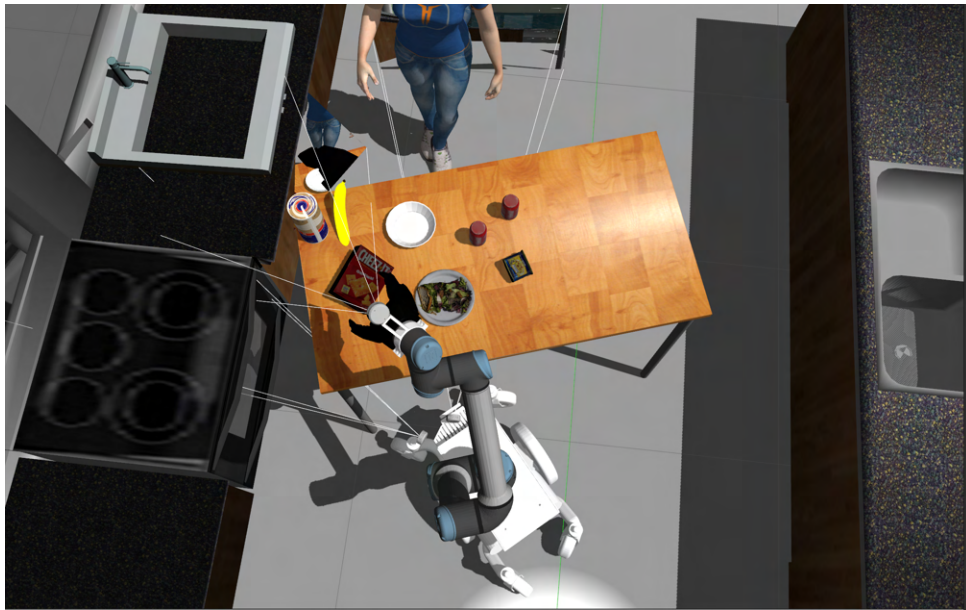


FIGURE 1.3: Amiga spawned in our final simulated environment with rich kitchen contexts such as humans, stoves, sinks, cans, bowls, salad and etc

The challenges associated with this project are extensive but can be laid out from two perspectives:

Module challenges: this project requires the designer to draw on multiple different fields: simulation, perception, manipulation, grasping and navigation. Each of these fields represents a distinct area of research. Moreover, since the robot is custom-assembled, various hardware components need to

be explored and integrated into the system in a compatible and portable fashion. Still, the particular challenge is to obtain an integrated system that can combine a variety of hardware and software components to increase the range of tasks the robot can perform while decreasing its dependence on prior information and increasing its awareness of its environment.

Engineering challenges: moreover, much of the challenge during the project lies engineering-wisely. In general, resources and documentation are very limited in this area. Working with each component is not as straightforward and often involves unexpected bugs and inconsistencies. Causes and solutions will be discussed in the relevant component implementation section in the hope of providing a good future reference. Configuration of the development environment also requires a considerable amount of effort, as it frequently involves up to five computers (a robot laptop, a user laptop, an embedded device and two onboard GPUs). The wiring and hardware components of the actual robot often fail unexpectedly, making work on them equally challenging. A significant amount of time was spent resolving these issues, which often resulted in a slowdown in progress.

Many utility scripts need to be developed and bugs fixed whose values cannot be showcased directly. Further, we will inevitably have to deal with more than 50 significant software packages and libraries in order to complete the final system implementation.

1.3 Contributions

The main contributions of this project can be summarised as follows:

1. A general framework architecture as in Figure 4.2 and 4.3 consisting of five sub-architecture which allows Amiga to move around and perform some kitchen tasks such as grasping and handling, mixing and stirring, pouring and more.
2. Amiga simulation framework as summarised in Figure 5.3, including a whole simulation of Amiga involving both the arm, gripper and the wheelbase, optimised calibrated camera plugin, grasp plugin and LiDAR plugin to allow for simulating perception, manipulation, grasping and navigation in the simulated kitchen scene. The module integrates smoothly with other modules.
3. Amiga perception framework as summarised in Figure 6.3, including a trained-from-scratch Kitchen object detection neuron network covering 37 classes and 100K images, running at 200 fps, a plane detection node which segments all planes with fps at least 15 fps and various self-proposed algorithms such as region detection and dimension estimation.
4. Amiga manipulation framework as summarised in Figure 7.3, which provides useful services such as dynamic obstacle avoidance, adjusting viewpoints, grasping and placing, mixing, pouring and more.

5. Amiga grasping framework as summarised in Figure 8.3, via replicating Grasp Pose Detection and Dex-Net results, allows Amiga to grasp objects with different gripper grasp type ranging from fruits to bowls, teaspoons, clothes, knives and much more.
6. Amiga grasping framework as summarised in Figure 9.3, via investigating different strategies in sensor sources, odometry sources and mapping techniques, allows Amiga to navigate autonomously in a kitchen environment.

1.4 Outline of Contents

Featured below is a brief overview of this report's structure and content. The modular nature of this project rationalises the presentation of background knowledge, design alternatives and summaries within the individual component chapters. Nonetheless, a brief prevision of general prerequisites and a system overview will be provided beforehand.

- ➔ **Chapter 2** lays the foundation for future discussions by introducing the necessary prior knowledge.
- ➔ **Chapter 3** introduces the actual working platform, Amiga, describing the mechanics of the hardware and middleware employed and explaining the reasoning behind the choices.
- ➔ **Chapter 4** introduces the final system designed by giving an architecture overview and a typical workflow.
- ➔ **Chapter 5, 6, 7, 8 and 9** present the background, implementation, architecture summary of the sub-architecture concerning simulation, perception, manipulation, grasping and navigation.
- ➔ **Chapter 10** test and evaluates some of the interesting aspects in the system since most of them are already covered in the relevant implementation chapters.
- ➔ **Chapter 11** concludes the project and draw attentions for future work.
- ➔ **Chapter 12** offers the instructions to launch this project.

2 Preliminary

This chapter presents a brief review of some essential prior knowledge.

2.1 URDF

For robotics simulation, it is vital to know the physical properties of the robot. These are usually depicted by a Unified Robot Description Format (URDF) file, a popular XML specification used in academia and industry to model multibody systems and their workspace. Robots in the specifications are composed of rigid links connected by joints; flexible parts are not considered [urdf, 2012]. As shown in Figure 2.1, any other rigid object in the world can be described as a link. The specifications within a link address the kinematics/dynamics, visual representations and collision models as shown below in Figure 2.2. Further, a robot's *end-effector link* is the link to which tools are attached in order to interact with the environment. An example is the right-most link in Figure 2.1's robot arm, on which a suction gripper is attached.

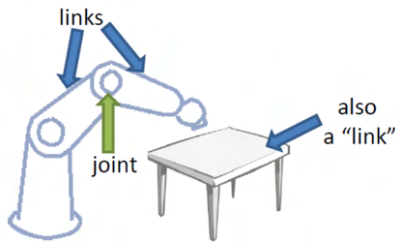


FIGURE 2.1: Links and joints specified in a URDF, extending to all rigid bodies

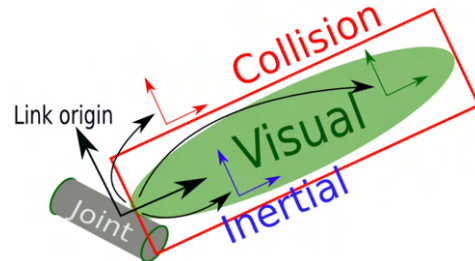


FIGURE 2.2: Origin, inertial, visual and collision information specified in a link attach to a joint

2.2 Rotation and Transformation

An essential foundation of robotics is to geometrically specify how a robot, or a rigid body, is positioned and oriented to a fixed reference coordinate system. While One can use the *Cartesian system* to describe positional translation, comparing orientations may not be as straightforward. There are two popular definitions for this cause, the Euler angle system and the quaternion system. The Euler angle is devised by Leonhard Euler and is defined as:

$$f_{eu} = (\phi, \theta, \psi)$$

where ϕ , θ , ψ are conventionally referred to as **roll**, **pitch**, **yaw**. As indicated in Figure 2.3 where red represents the x-axis, green represents the y-axis, and blue represents the z-axis, the Euler angles specifies three rotations, each representable by a *unitary matrix*:

A rotation from y-axis to z-axis around x-axis:

$$\mathbf{R}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \quad (2.1)$$

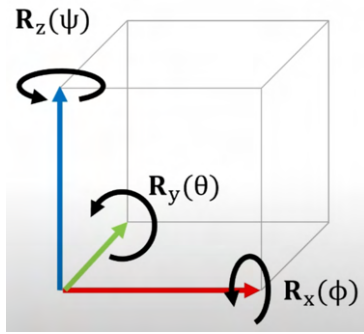


FIGURE 2.3:
Conventional
representation
of Euler angles

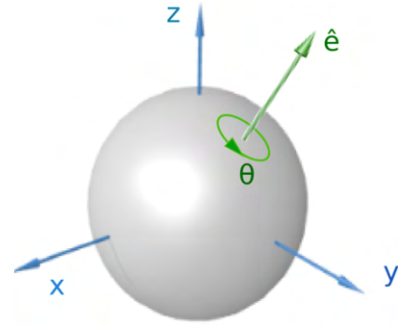


FIGURE 2.4:
Visualisation of
the quaternion
system as hyper-
sphere

A rotation from x-axis to z-axis around y-axis:

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad (2.2)$$

A rotation from x-axis to y-axis around z-axis:

$$\mathbf{R}_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.3)$$

Observe how the rows and columns associated with each axis of rotation are all zeros. Multiplier these matrices is sufficient to obtain a rotation around multiple axes – a generic rotation matrix, $\mathbf{R} \in \text{SO}(3)$ ¹, can be represented as the product of these rotation matrices:

¹3D rotation group

$$\mathbf{R} = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)^2 \quad (2.4)$$

$$= \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}$$

On the other hand, the concept of quaternions was introduced by William Hamilton in 1843 and essentially extended complex numbers to **four** dimensions. One real dimension and three imaginary dimensions make up each quaternion. This representation form is widely employed in various engineering systems primarily due to two reasons: it avoids the *gimbal lock*³ scenario, and it is computationally more efficient.

As shown in Figure 2.4, a quaternion consists of a vector part $\hat{e} = (x, y, z)$, $\hat{e} \in \mathbb{R}^3$ and a scalar part $\theta \in \mathbb{R}$:

$$f_q = (\theta, (x, y, z))$$

Or in complex form:

$$f_q = \theta + ix + jy + kz$$

where **i**, **j**, **k** are imaginary numbers. We will not go into too much detail about quaternions as it can get messy, and will state without proof the following relationship:

$$\mathbf{R} = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - \theta z) & 2(xz + \theta y) \\ 2(xy + \theta z) & 1 - 2(x^2 + z^2) & 2(yz - \theta x) \\ 2(xz - \theta y) & 2(yz + \theta x) & 1 - 2(x^2 + y^2) \end{bmatrix} \quad (2.5)$$

Note that Equation 2.4 and Equation 2.5 render the conversion between the two systems possible.

Further, let $\{A\}\mathbf{T}_P$ represents the translation⁴ of a point $P = (x, y, z)$ in frame $\{A\}$, the translation of P in frame $\{B\}$ is as follows:

$$\{A\}\mathbf{t}^P = \{A\}\mathbf{R}_{\{B\}\{B\}}\mathbf{t}^P + \{A\}\mathbf{t}_{\{B\}} \quad (2.6)$$

where $\{A\}\mathbf{R}_{\{B\}}$ defines the rotation from frame $\{A\}$ to $\{B\}$.

²Multiplication does not commute

³The loss of one degree of freedom in 3D

⁴A translation of a point to a frame is just the position with respect that frame

Alternatively, in a matrix form:

$$\begin{bmatrix} X_{\{A\}} \\ Y_{\{A\}} \\ Z_{\{A\}} \\ 1 \end{bmatrix} = \begin{bmatrix} \{A\}\mathbf{R}_{\{B\}} & \{A\}\mathbf{t}_{\{B\}} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_{\{B\}} \\ Y_{\{B\}} \\ Z_{\{B\}} \\ 1 \end{bmatrix} \quad (2.7)$$

The 4x4 matrix in Equation 2.7 is sometimes referred to as the *homogeneous matrix*; an example can be found in Equation 2.12.

Remark

To conclude, we established the transformations between frames of a 3D point $P = (x, y, z)$. It will be more sophisticated to transform a 6D pose, which contains a 3D position and a 3D location. Relevant discussions will be continued in the implementation sections under the specific project settings. In this project, we employ the ROS *tf* library [tf, 2017] to maintain the transformations amongst various coordinate frames in a robot. The *tf* library provides convenient methods to broadcast, retrieve, visualise and perform transformations. However, a clear understanding of the concepts behind rotations and transformations plays a crucial role in implementing the motion planning and grasping modules; this is particularly true for the 6DoF grasping module as introduced in Section 8.1.3.

2.3 Camera Model

Cameras are optical instruments used to capture visual images. A pinhole camera model maps 3D points in world coordinates into 2D points in the image plane through the following projective transformation equation [Hartley and Zisserman, 2004]:

$$\begin{bmatrix} kx_i \\ ky_i \\ k \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (2.8)$$

where the point $[kx_i, ky_i, k]$ and $[x, y, z, 1]$ represents the *homogeneous coordinates* of the image plane and the real world, respectively.

Figure 2.5 illustrates the concept of a pinhole camera model, where $P_w = (X_w, Y_w, Z_w)$ represents a 3D point expressed with respect to the world coordinate system, $P_c = (X_c, Y_c, Z_c)$ with respect to the camera coordinate system, $p = (u, v)$ represents a 2D pixel in the image plane, and (c_x, c_y) the principal point, where the perspective center is projected.

The 2D pixel and the 3D point in the world coordinate system are related by the equation:

$$p = KP_c \quad (2.9)$$

where K stands for the camera intrinsic matrix.

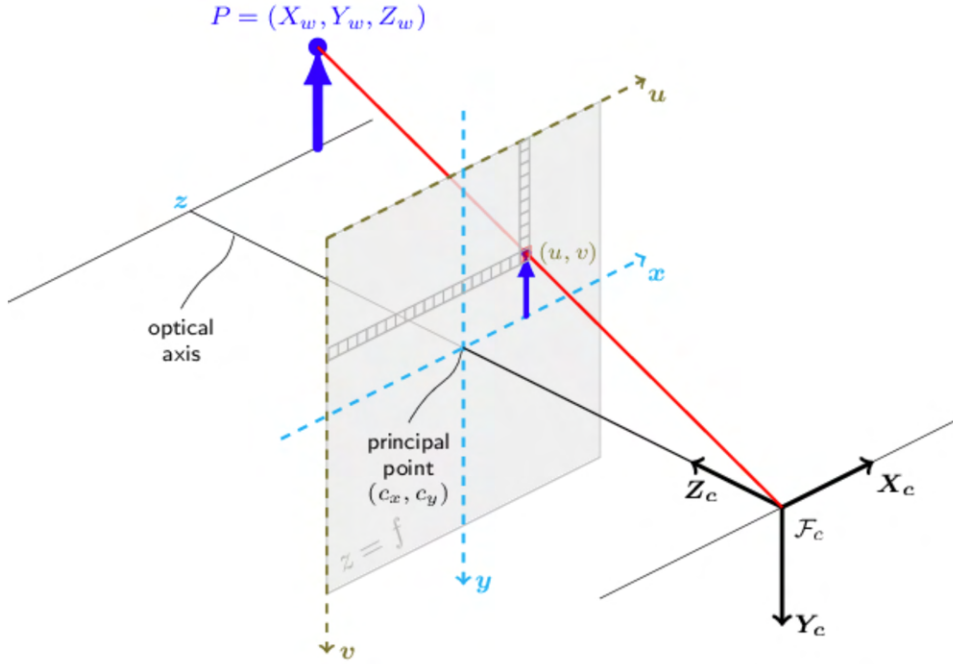


FIGURE 2.5: Pinhole camera model [OpenCV, 2000]. The 3D point in the world coordinate system is mapped to the 2D pixel in the image plane. $\mathcal{F}_c$ is the camera coordinate system.

where the intrinsic matrix K is composed of the focal lengths (f_x, f_y) , the principal point (c_x, c_y) , and axis skew s :

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.10)$$

There is no dependency between the matrix of intrinsic parameters and the scene viewed. Therefore, it can be re-used once estimated, provided the focal length remains unchanged. This implies that if an image from the camera is scaled by a factor k , all of the parameters need to be scaled by the same factor [OpenCV, 2000]. Given this knowledge, combining Equation 2.9 and Equation 2.10 yields:

$$k \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \quad (2.11)$$

Further, the augmented position vector of a 3D point in the camera frame and the world frame is related by a *homogeneous transformation*:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad (2.12)$$

where $[r_{11} \dots r_{33}]$ spans a rotation matrix and $[t_x \dots t_z]$ spans a translation matrix. Finally, combining Equation 2.11 and Equation 2.12 yields the mapping between the 2D pixel and the 3D point in the world coordinate frame:

$$k \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad (2.13)$$

Note that multiplying the 3x3 intrinsic matrix with the 4x3 world-to-camera transformation matrix results in the camera matrix as in Equation 2.8.

Note that the above analysis does not account for camera distortions, which may include *chromatic aberration*, *spherical aberration* and *radial distortion* (most commonly seen in real lenses).

Moreover, the process of estimating values C_{ij} is known as *camera calibration*, where many correspondences of known coordinates between the world and camera frame are established to produce a *homogenous linear system*:

$$\mathcal{P} \begin{bmatrix} C_{11} & C_{12} & C_{13} & \dots & C_{34} \end{bmatrix}^T = 0$$

which could be solved by a direct linear transformation algorithm via *SVD decomposition* [Mikolajczyk, 2022].

2.4 Depth Map and Point Cloud

The depth map and the point cloud are two of the most common data types provided by RGB-D cameras in 3D computer graphics and computer vision. Conversions between these two data types are possible provided the camera matrix is known, which is explained in detail already in the last section.

A depth map is an image channel that captures information relating to the distance to the surfaces of scene objects from a particular viewpoint. Each pixel location is associated with a numerical value indicating the depth. The depth maps are corrected to align with RGB images pixel-by-pixel through a process called *depth image rectification*, which is often carried out by the manufacturers in advance.

On the other hand, a point cloud represents a 3D shape or object as a collection of data points in space. Each point position is associated with a set of Cartesian coordinates (X, Y, Z). Efficient processing of 3D point clouds is a vivid area of research in image processing, giving rise to numerous algorithms such as filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. Point Cloud Library (PCL) [Rusu and Cousins, 2011] and Open3D [Zhou, Park, and Koltun, 2018] are two open-source libraries that support rapid development of software dealing with 3D data and will be the essential dependency of this project in regards to processing point clouds. Also, it should be noted that point clouds contain far more information than simply positions, such as intensities and curvatures, resulting in a relatively large size..

Remark

PCL is chosen for this project because it has a long history of stable releases and a great ecosystem since 2011. However, the Python wrapper for PCL [pythonPCL, 2018] is no longer maintained after 2018. As a backup, Open3D is used for rapid prototyping, debugging and powerful visualisation in Python.

2.5 RANSAC

The random sample consensus (RANSAC) [Fischler and Bolles, 1981] is an algorithm for robust fitting of models in the presence of many data outliers.

RANSAC may be thought of as a combined process of two iterative steps, as illustrated in Figure 2.6:

- Randomly choose a subset of samples from the data. By using these data, a fitted model and associated model parameters are obtained. The number of sample subsets is minimally sufficient to obtain the model parameters⁵.
- Check which elements of the Dataset are consistently in the model estimated in the first step. If the model element does not fit the estimated model outside the threshold (maximum deviation from the noise), it is treated as an outlier.

The setting of inliers is called the "consistency setting", and the RANSAC algorithm iterates until enough inliers are obtained. The step-by-step procedure can be summarised as follows:

1. Randomly select a subset of the original data, i.e. the hypothetical inliers
2. Build a model fit
3. Validate with other data, calculating whether the model fits according to the model-specific loss function

⁵As far as a plane model is concerned, at least three parameters are required

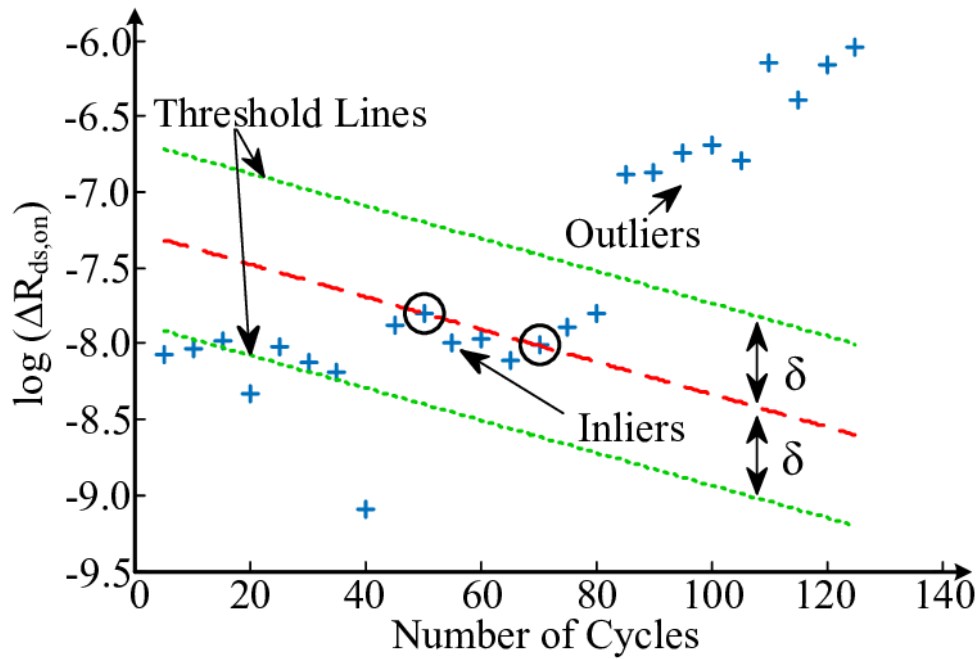


FIGURE 2.6: Illustration of the RANSAC algorithm to detect outliers [Heydarzadeh et al., 2017]. Errors larger than the threshold lines will be regarded as outliers

4. If enough points are considered to be in the "consistent" setting, then the model is considered to be a good model
5. Comparing all the "consistency" settings (i.e. all the models constructed) to determine which inliers are most desirable

In each successive application of this procedure, a refinement of the model will result in an error or a rejected model because there are only a few data points classified as inliers. If the error of the refined model is lower than that of the last saved model, it will be retained.

The RANSAC algorithm has the following benefits:

- Simple and general
- May be applied to a variety of problems and is experimentally proven successful in many practical scenario
- Resilient to a large number of outliers ($\sim 50\%$)
- Easily adjustable parameters (number of sample subset, number of iterations and distance threshold)

and the following drawbacks:

- Computation proliferates as the number of model parameters grows
- Can be problematic when models are approximate

2.6 Deep Learning and CNNs

Deep learning is a branch of machine learning methods that incorporates artificial neural networks in conjunction with representation learning. It can be supervised, semi-supervised, or unsupervised [LeCun, Bengio, and Hinton, 2015]. *Convolutional neural networks (CNNs)* are artificial neural networks that are primarily utilized for visual image analysis [Simos, Katsikis, and Mourtas, 2022]. CNNs provide translation-equivariant responses (also known as feature maps) by applying a shared-weight architecture to their convolution kernels, which have proven to be extremely effective for processing locally-correlated pixel data.

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [Deng et al., 2009] is a standard benchmark for comparing image recognition systems at a large scale. It contains more than 1.2 million images across a wider variety of objects with 1000 classes for recognition. Annual competitions at ILSVRC have been held since 2010 and have continued to drive CNN architecture's progress as depicted in Figure 2.7, where the horizontal axis indicates model complexity, the vertical axis represents accuracy in the ImageNet Dataset, and the magnitude of the circle reflects the number of parameters.

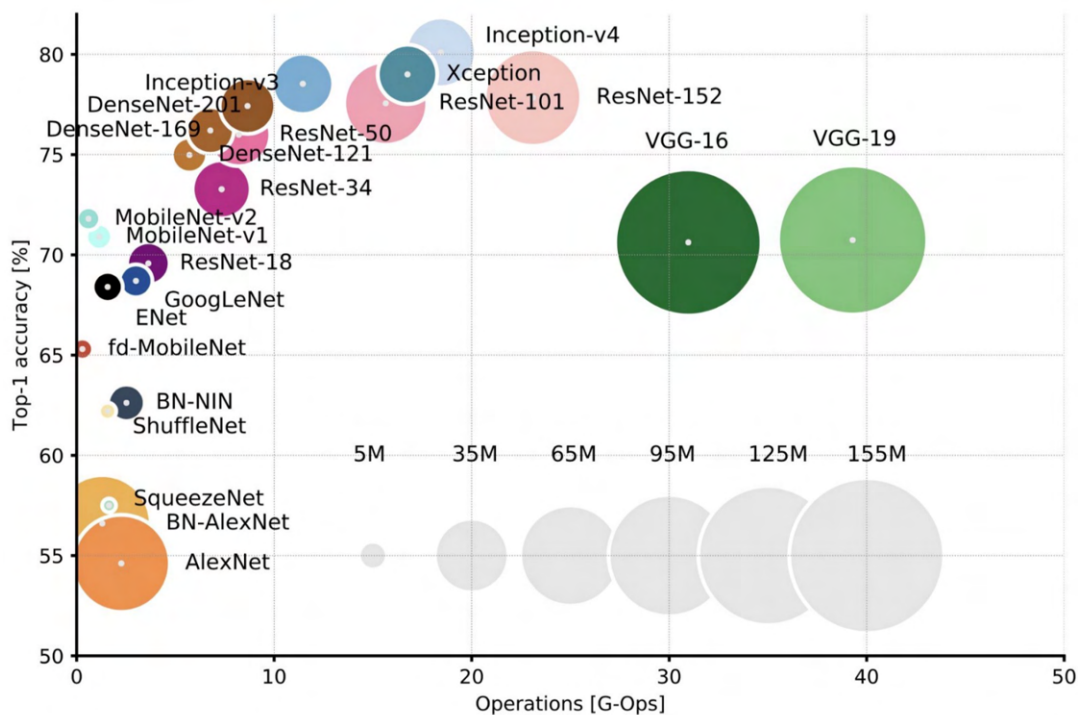


FIGURE 2.7: Performance comparison of various CNN architectures up to 2018 [Mikolajczyk, 2021]. The horizontal axis indicates model complexity, the vertical axis represents accuracy in the ImageNet Dataset, and the magnitude of the circle reflects the number of parameters

The standard process of applying deep learning to a specific task (for example, the recognition of kitchen scene objects) involves selecting a CNN

backbone architecture that has been pre-trained in ImageNet and fine-tunes it locally; this is known as *transfer learning*.

Remark

With regard to this project, a lightweight model is preferred if possible because robotics manipulation often requires real-time feedback from the scene. Model inference latency and data transmission latency constitute the overall feedback latency. While the former can be resolved using Ethernet cables, we prefer a lightweight architecture with excellent real-time performance, such as the MobileNet series from Figure 2.7.

2.7 Wheel Odometry

For mobile robots with two motors (one per wheel)⁶, steering can be achieved by setting different speeds.

1. For straight-line motion, the wheels move at the same speed.
2. During a turn, both wheels operate at the same speed but in the opposite direction.

Let the velocities of the left and right wheels be V_L and V_R , the distance between them be W , and supposes a period of motion Δt where the robot moves along a circular arc of radius R through angle $\Delta\theta$ as shown in Figure 2.8. We have:

$$\Delta\theta = \frac{V_L \Delta t}{R - \frac{W}{2}} = \frac{V_R \Delta t}{R + \frac{W}{2}} \quad (2.14)$$

and hence:

$$R = \frac{W(V_R + V_L)}{2(V_R - V_L)} \quad (2.15)$$

$$\Delta\theta = \frac{(V_R - V_L)\Delta t}{W} \quad (2.16)$$

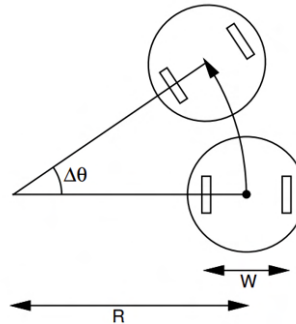


FIGURE 2.8: Circular path of a differential drive robot

Equations 2.15 and 2.16 set the basis for wheel odometry.

⁶Although Amiga has six wheels, only one per wheel has motors

3 Introduction to Amiga

3.1 Amiga and Hardware

Amiga is a customarily-assembled mobile manipulation robot at the personal robotics lab, Imperial College London. The robot can be divided into five core parts, RGB-D/Lidar cameras and Laser scanners for sensors, arm-grippers for manipulation, a wheelchair base for navigation, laptops and embedded computers for computation and routers/controllers for networking.

Amiga's hardware adapts to changes throughout the project. Figure 3.1 shows a set of images of Amiga taken at different stages during the project. A summary of all the hardware utilised in each core part is as below:

- **Sensing:** Intel RealSense L515 and T265 Cameras, ZED 2 Stereo Cameras, RPLIDAR A1M8 Laser Scanner
- **Manipulation:** UR10e Collaborative Robot Arm with Tablet, Robotiq 3-Finger Adaptive Gripper
- **Navigation:** Quickie Electric Wheelchair Base, R-NET System Modules
- **Computation:** Laptop with RTX 3070 GPU, Nvidia Jetson Xavier NXs, Raspberry Pi 4
- **Networking:** Routers, Switches, Emergency Stops

Figure 3.2 presents a front view image of Amiga with its hardware labelled. Each core hardware component and its design choice will be further elaborated in the following subsections.

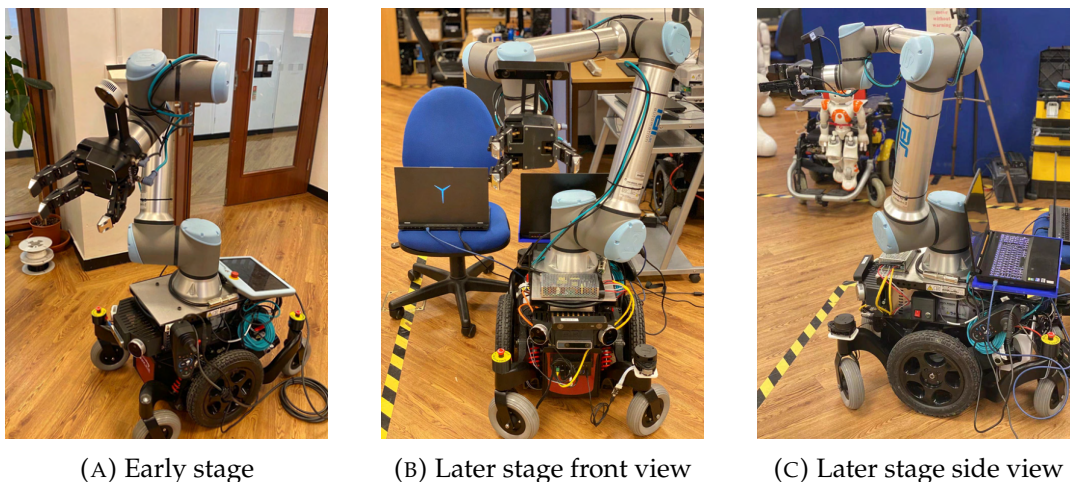


FIGURE 3.1: A set of images of Amiga taken at different stages

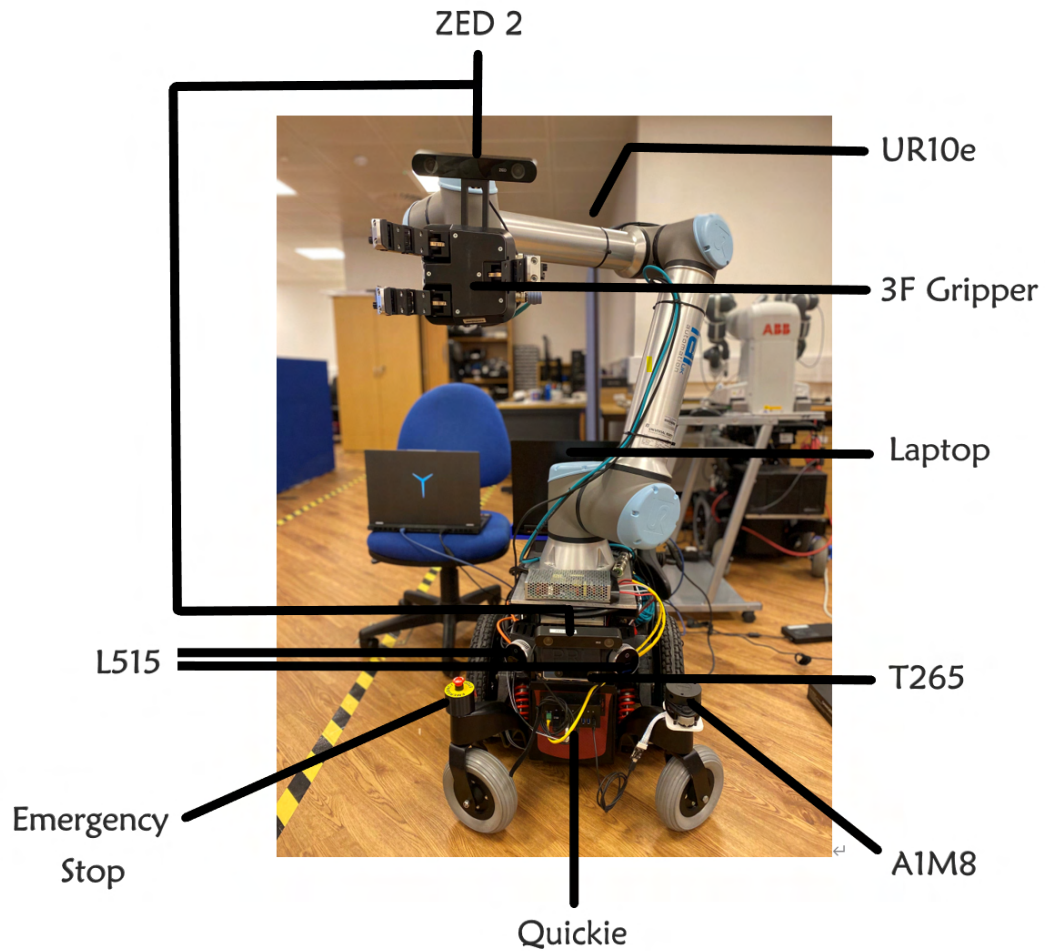


FIGURE 3.2: Amiga hardware components labelled (front view)

3.1.1 Cameras and Laser Scanners

RGB-D cameras: RGB-D cameras are depth-sensing devices that associate traditional RGB images with depth information. Two common types of examples are LiDAR and stereo cameras. In a nutshell, stereo cameras simulate human binocular vision by comparing two images from two sensors of a known distance apart; and LiDAR cameras calculate depth by evaluating the time taken for light to travel forth and back.

There are, in total, five cameras mounted in Amiga. At the early stage of the project and as indicated in Figure 3.1a, an Intel RealSense L515 LiDAR Camera (Figure 3.3a) was mounted on the wrist of the arm via a 3D printed clamp; two more L515 cameras were mounted on the front of the wheelchair base, along with another RealSense T265 Tracking Camera (Figure 3.3b). Moreover, to increase the range of FoV¹ seen by the wheelchair base, the two L515 cameras are mounted tilted left and right, with the T265 camera positioned in the centre.

¹Field of View (FoV) depicts the angular extent of a given scene imaged by the camera



FIGURE 3.3: RGB-D cameras used in Amiga

Alternative camera options exist. In the beginning, the L515 LiDAR cameras were selected because they have a reasonably high 1920×1080 resolution given their lightweight nature, which allowed for easier mounting. This was particularly true for the wheelchair base, for which there were fewer options due to the limited accommodation space on the 3D printed camera mount as visible in Figure 3.4 below:

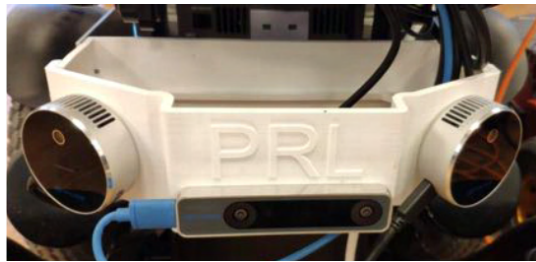


FIGURE 3.4: 3D-printed camera mount hosting two L515 and one T265 cameras. The spaces are quite limited

The T265 camera was used because it had two wide Fisheye lenses with a combined $163 \pm 5^\circ$ FOV, making it suitable for Visual SLAM applications.

Furthermore, as the project developed, it was discovered that the L515 camera mounted on the arm was experiencing severe **overheating** and **disconnection**. Since the arm was extended, an equally extensive USB cable was required to connect the camera all the way to the embedded computers located at the centre of the wheelchair base. However, lightweight Lidar cameras such as the L515 could not deal with this situation properly. Also, Intel announced in February 2022 that it would cease to produce this type of cameras. As a result, another two stereo cameras ZED 2 (Figure 3.3c), were used on the arm and on the wheelchair base instead. They had a better resolution of 2208×1242 and were much more stable under our setup, able to run continuously for over six hours uninterrupted.

A quick recap of the current camera setup can be found again in Figure 3.2. The cameras are flexible for different tasks. For the moment, the ZED 2 camera on the arm is used for manipulation and the others for navigation.



FIGURE 3.5: Laser scanner, arm and gripper in Amiga

Laser Scanners: Although many cameras were used for navigation, sensing was still restricted to the front. To address this problem, two RPLIDAR A1M8 Laser Scanners (Figure 3.5a) were attached to the left-front and right-back host adaptors of the wheelchair base. These scanners allowed for 360° complete range scanning in a radius of twelve metres.

3.1.2 Robotic Arm and Gripper

In the context of manipulation, a robotiq 3 finger gripper (Figure 3.5c) is attached to the end-effector of a UR10e robot arm (Figure 3.5b). As mentioned previously, a camera is installed on the end of the arm and enables Amiga to adjust the viewpoint to retrieve visual information about the scene accordingly. The arm can be controlled by a tablet which is provided by the UR manufacturer. In addition to the firmware and control interface, the tablet will come with built-in features such as a teach-and-repeat programming mode, with which the arm is shown the actions by a human demonstrator and a series of waypoints and control signals are stored to repeat the movements on demand. Nevertheless, for the purposes of this project, these context-dependent functions will not be further explored.

Arm: While Amiga uses a wheelchair base, it is not designed to function as an assistive wheelchair robot². Therefore, the ample space allows it to accommodate an equally large arm. The UR10e is a universal collaborative industrial robot arm with 6 degrees of freedom. It features both a high payload lift of 12.5 kg and a long reach of 13 metres and hence is an excellent choice over lightweight robotic arms such as the Kinova Gen3 [kinova, 2020].

Gripper: Still, a variety of grippers could couple with UR10e, most of which are in the form of parallel jaws and hence could only support one type of grasps. By contrast, the 3-Finger Adaptive Gripper is a versatile gripper with force, position and speed control for each finger and different grip modes that can almost pick anything. Figure 3.6 demonstrate a set of grasps the 3-F gripper can perform. Moreover, besides the basic mode, the pinch mode and

²An assistive wheelchair robot is designed to interact with people seated on it

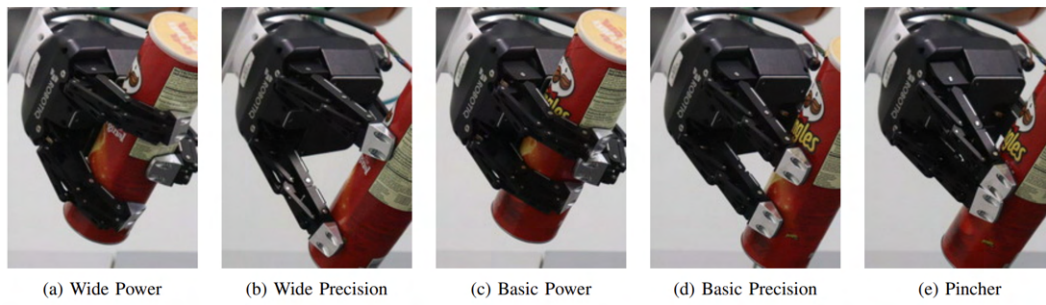


FIGURE 3.6: An illustration of five types of grasps that can be achieved with the Robotiq 3-Finger Adaptive Gripper [Corsaro, Tellex, and Konidaris, 2021]. Different type has its own strength in grasping different objects. The scissor type is not shown here

the wide mode shown in the figure, it also has a scissor mode, which utilises the two fingers from the same side to operate like a scissor. It is envisaged that the dexterity of this gripper will be examined as part of the project scope.

3.1.3 Wheelchair Base and R-NET

The navigation aspect of Amiga consists of a Quickie wheelchair base (Figure 3.7a), a R-NET joystick (Figure 3.7b) and a power module (Figure 3.7c).



(A) Sunrise Quickie Salsa-M2 [Quickie, 2022]



(B) R-NET joystick module [Joystick, 2022]



(C) R-NET EL power module [Power, 2022]

FIGURE 3.7: Wheelchair base, R-Net joystick and power module used in Amiga. The wheelchair's seat has been removed

Wheelchair base: Designed to handle a wide range of usages, the Quickie wheelchair base is a highly-specified electric wheelchair. Several assistive wheelchair robots at the personal robotics laboratory have equipped this base and proven successful, rendering it the ideal choice for Amiga. With mid-wheel drive, the wheelchair base can ensure excellent traction and seamless rotation on the spot. The six wheels are independently suspended, providing a smooth and stable ride. The top speed can be set up to six mph³.

³Miles per hour

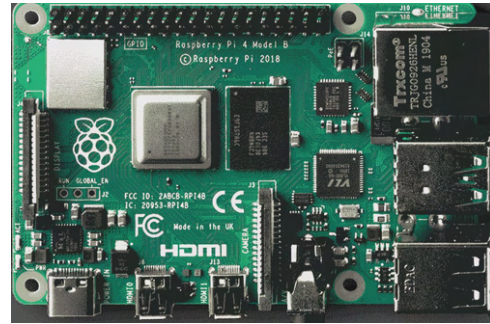
R-NET: The R-Net system [R-Net_System, 2017] is a multi-module control system for rehab powerchairs such as Quickie. The system is highly flexible and adaptable, allowing up to fifteen modules to be connected. There are joystick modules, power modules, intelligent seating/lighting modules, and advanced stability modules, among others. The integration of modules depends on the application's requirements, with no restrictions on compatibility, and modules can be mixed and matched to provide the best performance over cost. Amiga adopts the joystick and power modules as part of its navigation system. Besides providing user-friendly base control, the joystick also features a convenient, industry-standard charging port, which is utilised along with the EL power module for charging Amiga in its entirety. Moreover, two inhibit inputs are available on the EL Power Module, allowing the programming of varying levels of speed.

3.1.4 Embedded Edge Computers

It is essential to have computers in order to run software and algorithms in Amiga. Nevertheless, connecting multiple laptops is not feasible because of insufficient space; and cloud computing renders the system non-real-time. The solution is to employ embedded edge computers that contain a Broadcom System on a Chip (SoC)⁴. Apart from a laptop, Amiga has used two Jetson Xavier NXs (Figure 3.8a) and one Raspberry Pi 4 (Figure 3.8b).



(A) Jetson Xavier NX [NX, 2020]



(B) Raspberry Pi 4 [Pi, 2019]

FIGURE 3.8: Embedded edge computers used in Amiga

Jetson Xavier NX: Featuring a low power consumption design, Jetson is an ideal edge platform for accelerating machine learning applications. Among other things, Jetson NX has elevated supercomputer performance through the ability to run modern neural networks in parallel and process data from multiple high-resolution sensors simultaneously. Amiga utilises two of these modules to run computation-intensive vision algorithms for the arm and base which frequently employ convolutional neural networks. These boards are connected to other hardware components via Ethernet cables.

⁴A SoC often features an integrated central processing unit (CPU) and on-chip graphics processing unit (GPU)

Raspberry Pi 4: Generally speaking, Raspberry Pi 4 is similar to Jetson, except it has a less capable GPU. Amiga uses this module as a bridge between the local hardware network and the R-NET navigation system, which typically does not permit external methods of communication to the wheel motors other than its own modules.

3.1.5 Controllers and Networking

Amiga utilises a router to establish a local communication network amongst various computers described in the previous subsections. The network assigns each computing device a unique IP address. It permits both Ethernet and Wi-Fi communications, although the latter carries a significant latency and is therefore unsuitable for large-scale data transmissions. An example of an external user's laptop connecting to Amiga's network through an Ethernet cable can be found in Figure 3.2. Moreover, A graphical illustration of Amiga's current networking is provided in Figure 3.9 as below.

Furthermore, switches and controllers are incorporated into the power circuit, as in most electric systems. Two emergency stops are mounted on both the left-front and right-back ends of the wheelchair base's host adaptors so that users may intervene during Amiga's movement at any time.

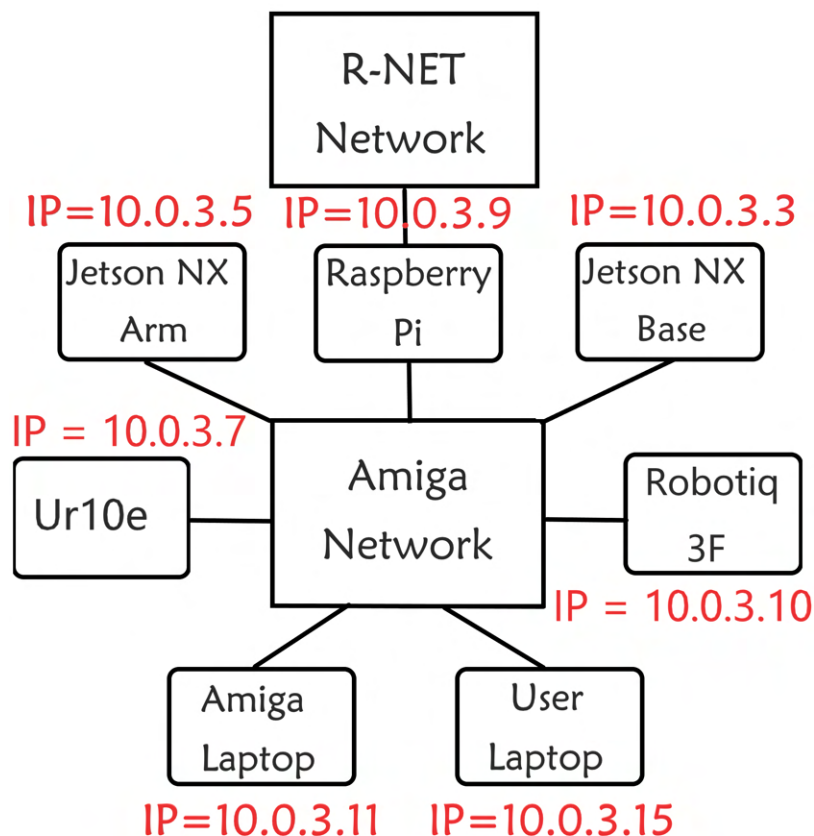


FIGURE 3.9: Graphical illustration of Amiga's networking. Each hardware is associated with a unique IP address

3.2 ROS and Middleware

ROS is a collection of open-source software libraries and tools to facilitate the development of robotics applications [ROS, 2021]. Moreover, ROS is available in a wide range of features, from low-level device control, hardware abstraction and message communication to high-level application functions, development tools and package management. Ever since its creation by Stanford Artificial Intelligence Laboratory in 2007, ROS has grown and evolved into the dominant robotic middleware that supports a vibrant community and ecosystem. In order to integrate closely with other state-of-the-art robotic applications, Amiga leverages ROS as the middleware system.

Figure 3.10 provides a visual representation of ROS's core concepts. Figure 3.10 depicts roscore, which adopts ROS Master, ROS Parameter Server and rosout. ROS Master provides node identification and registration, assigning unique pids⁵ to nodes. ROS Parameter Server manages and controls the overall parameters in ROS. Rosout monitors and reports the console logs.

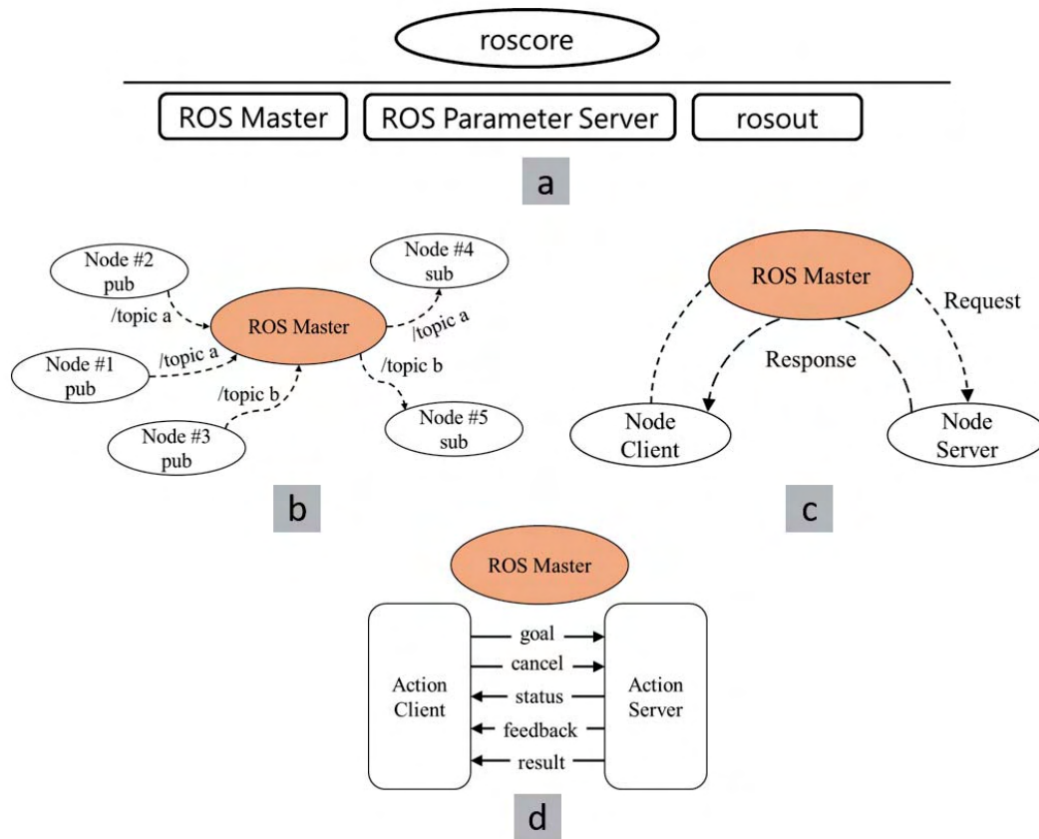


FIGURE 3.10: An illustration of ROS graph concepts [Deng, Xiong, and Xia, 2017]. The core concepts are the ROS Master, the nodes and the publisher-subscriber / client-server models

⁵Process IDs

A ROS node is a software module. ROS nodes can locate each other and establish peer-to-peer communications with three mechanisms [Deng, Xiong, and Xia, 2017], which are illustrated in Figure 3.10 b - d. These mechanisms are summarised as follows:

- **Publisher - Subscriber:** asynchronous one-way communication mechanism for transferring messages amongst nodes (Figure 3.10 b)
- **Client - Server:** synchronous one-way with feedback communication mechanism that sends/responds to requests/data (Figure 3.10 c)
- **Action Client - Server:** similar to client/server mechanism but allow for bidirectional communication (Figure 3.10 d)

Based on their characteristics, topics and services have different use cases. Topics have many-to-many connections, low real-time performance and no feedback, rendering them ideal for message passing. In comparison, services have one-to-many node relationships, high real-time capacities with feedback, and are thus suitable for logic processing.

Furthermore, ROS provides a number of useful tools and packages that we will use quite frequently in this project:

- **rviz:** 3D visualization tool [Rviz, 2019]
- **tf:** Coordinate frames monitoring tool [TF, 2020]
- **rqt:** Qt-based framework for GUI development [rqt, 2016]
- **roslaunch:** tool for quickly launching multiple ROS nodes locally and remotely via SSH [roslaunch, 2019]
- **rodebug:** a collection of tools that allow for measuring average rate, generates tf trees and monitor specific transformation [rosDebug, 2017]

4 Analysis and System Implementation

In this chapter, we revise the project's requirements, analyse them and present our system implementation.

4.1 Requirement Capture and Analysis

Recall our requirement is to devise a robot cooking assistant system for Amiga. Such system must possess the following properties:

- *Generality* – the system should be capable of performing various tasks, acquiring new skills, applying these skills in novel situations, adapting continually, and improving its performance.
- *Extensibility* – the system should be designed with high cohesion and low coupling in order to provide a straightforward and effortless process for adding functions within the same framework.
- *Distributability* – the system should be portable and scalable to achieve optimal computation utilisation across different types of computers used in the robot (laptops, embedded devices, etc) with different configuration environments.

Remark

*The requirement of **extensibility** prompts us to design a **modular** system such that optimisations and expansions can be easily performed on a component basis. In addition, using ROS enables us to **distribute** modules across a network. Moreover, since disparate computing devices are used that run in different environments, we could Dockerise [Merkel, 2014] the modules to create a **portable** version. In relation to the generality requirement, since we are working towards a portable, modular architecture, the **generality** of each component would suffice to conclude the general nature of the system as a whole. A list of requirements for each module is as below:*

1. *Simulation Module* – Amiga should be associated with a simulation environment that allows modules to be tested offline and debugged quickly:
 - (a) It should be easy to build different kitchen environments
 - (b) It should be easy to integrate other modules with it
2. *Perception Module* – Amiga should possess a certain degree of knowledge in understanding the semantics of the kitchen environment:

- (a) It should be easy to recognise more kitchen objects
- (b) It should be easy to retrieve desired locations on recognised objects
- 3. *Manipulation Module* – Amiga should plan and execute arm movements in accordance with dynamic obstacles introduced by the mobile base:
 - (a) It should account for dynamic obstacle environments
 - (b) It should be easy to change the arm’s view for exploration
- 4. *Grasping Module* – Amiga should possess a certain degree of knowledge and capabilities in grasping a variety of common kitchen objects:
 - (a) It should try to reduce reliance on perception capabilities
 - (b) It should be easy to switch to different grasp types
- 5. *Navigation Module* – Amiga should be able to localise and navigate around the kitchen without hitting any obstructions:
 - (a) It should be easy to extend to new kitchen environments

Certainly, the list is not definitive, but rather a logical consequence of a system deemed general.

4.2 System Overview

In light of above analysis, we propose a modular and portable system (Figure 4.2) distributed over the Amiga network¹.

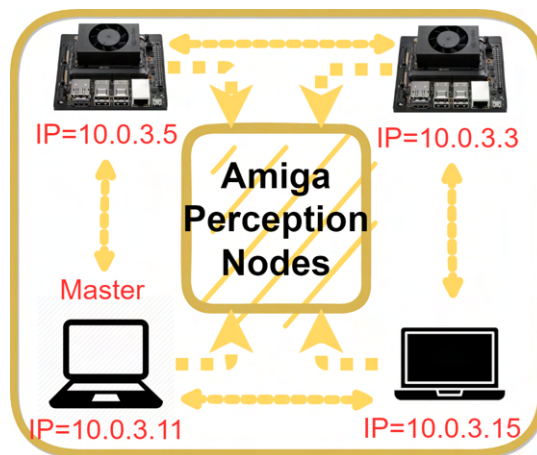


FIGURE 4.1: Inside Amiga perception: individual nodes are running on various computing devices, each with a unique IP address. When launching a node, each device must declare the Master’s IP as well as its own IP. Any device can be rendered as the master (in our case, the master is Amiga’s laptop)

¹The configuration of the Amiga network must be credited to Goubard Cédric

As shown in Figure 4.2 below, the system consists of five core modules: Amiga simulation, Amiga perception, Amiga manipulation, Amiga grasping and Amiga navigation.

Each module is a high-level abstraction of a collection of software nodes running at different computing devices. As an example, Figure 4.1 demonstrates the infrastructure of Amiga perception. Building on top of Section 3.1.5 and Section 3.2 from the last chapter, each computing device has a unique IP address and can be reached from any other device in the network through ROS, provided two environment variables, ROS_MASTER_URI and ROS_IP, are set. Moreover, ROS_MASTER_URI specifies the host² of the network and ROS_IP specifies the IP of the device launching the node.

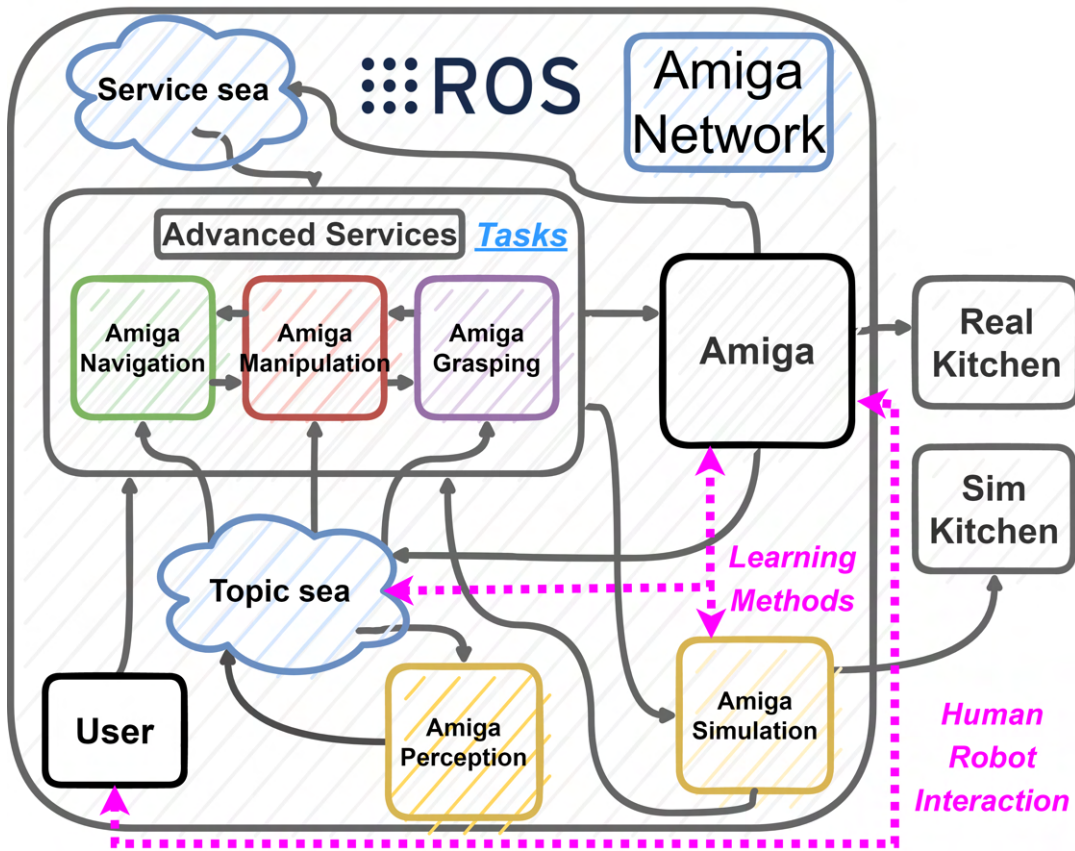


FIGURE 4.2: System overview. Within the Amiga network, various sensors and actuators of Amiga publish/advertise rudimentary topics/services to the topic/service sea. Amiga perception receives raw topics, processes them and publishes sophisticated ones for Amiga manipulation, grasping and navigation, which are essentially advanced services. A user within the network can call a service and invoke effects on the real kitchen via Amiga or the simulated kitchen via Amiga simulation. The pink dashed lines represent possible future work, which can include robot learning and human robot interaction

²In our case, the master runs in Amiga's laptop with IP = 10.0.3.11

To launch a node at a particular device, we put in a Makefile:

```
ssh -X ${USER}@${AMIGA_DEVICE_IP} "commands .."
docker exec -it amiga bash -c "roslaunch .."
```

Presently, we have more computing resources available than we need. As each module evolves, a future study of splitting computing power may be conducted.

As shown in Figure 4.2, our system divides into five components, Amiga simulation, Amiga manipulation, Amiga navigation and Amiga grasping. In the network, Amiga, essentially a collection of different hardware, broadcasts topics such as camera messages and advertises services such as gripper control to the topic/service sea. Amiga perception processes the raw vision topics and publishes sophisticated versions back to the topic sea. Amiga navigation, manipulation and grasping listen to these advanced topics and infer where/how to navigate with the wheels, where/how to manipulate with the arm and where/how to grasp with the gripper. Essentially, these three modules are also services, so a user could easily join the network (via Ethernet, wifi, etc.) and call them to generate plans, which are then sent to Amiga's motors and invoke influences on the kitchen world. The user could also influence the simulated kitchen world via Amiga simulation instead.

Furthermore, it is imperative to realise that the existence of Amiga manipulation and navigation implies that we independently control the base and arm. While there exist whole-body control methods based on *optimisation* [Berenson, Kuffner, and Choset, 2008] and *reinforcement learning* [Wang et al., 2020], reproduction and extension of these methods are difficult. As a result, we have essentially decomposed the mobile manipulation problem into two sub-problems of motion planning – the robot would drive to a particular location, park, and then manipulate³.

Any robot cooking assistant task can be viewed as a combination of navigation, manipulation and grasping problems. As shown in Figure 4.2, these tasks are wrapped as advanced services that a user may request at any time.

Remark

The system is expandable; two interesting directions for future work are shown in the pink dashed line in Figure 4.2:

1. **Human-Robot Interaction:** *Indeed, a robot cooking assistant needs to interact with humans in two directions, rather than just having a human call for an assistive service⁴. A separate module could be added between the Amiga and the user entity to accomplish this.*
2. **Robot learning:** *Instead of relying on the analytically programmed services to attain a task, we could also attempt learning methods such as reinforcement learning and imitation learning. For this purpose, we can connect the Amiga*

³Manipulators' control is typically much more accurate than navigators' counterpart

⁴Currently service calls are achieved via command lines but can be easily substituted with speech recognition, thanks to [ROS_Speech, 2021]

entity in the figure with the topic sea and the simulation module, bypassing the advanced service modules. Training can be conducted off-line in the simulation environment and the trajectory results applied to the real robot.

4.3 Data Workflow

Figure 4.3 illustrates an example of our system's data workflow in performing robot cooking assistance tasks. Getting to the bottom of the workflow directly can be difficult, so we will briefly run it through first, then disassemble and present each piece separately from scratch in the next few chapters.

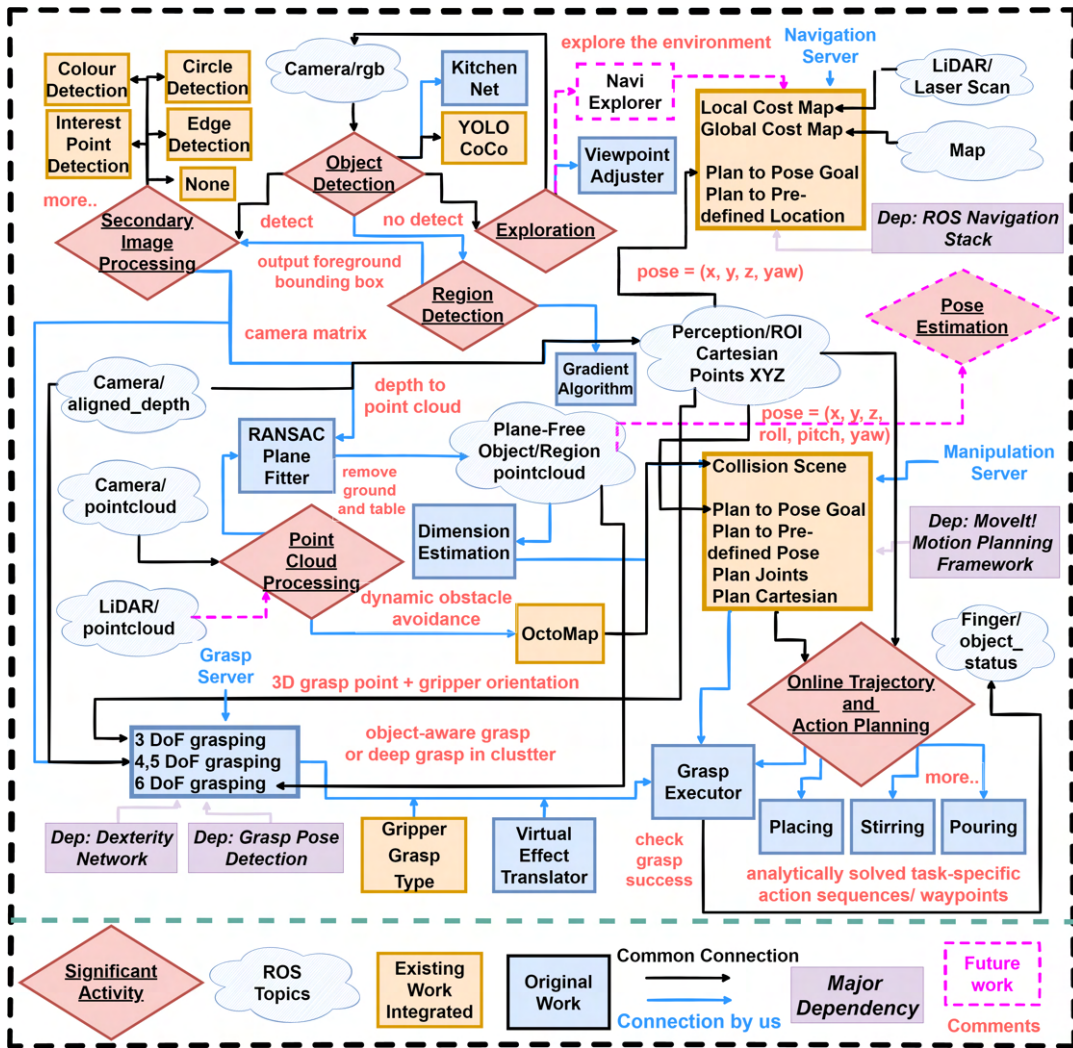


FIGURE 4.3: System workflow. Beneath the green dashed line are the meanings of each colour and symbol

Suppose Amiga is to grasp an object from the kitchen and hand it to the master. Starting from the object detection activity from Figure 4.3, if detection fails, we can:

1. enter an exploration stage until the object is detected

2. detect a foreground region of the cluster instead

Object and region detection results constitute a bounding box, which can be applied to secondary image processing to obtain more specific pixels in the region of interest. In turn, these pixels align with the depth map and produce the region-of-interest 3D coordinates.

These coordinates can be used as:

1. goals to the navigation server to plan and execute a path to a location close to the object⁵. Obstacle avoidance is guaranteed by the pre-built global cost map and the online local cost map from the LiDARs.
2. goals to the manipulation server to plan and execute a path such that the end-effector reaches the position⁶. Obstacle avoidance is guaranteed by the collision scenes.
3. 3D position of the grasp pose of the object; 3D orientation can be the current gripper pose

Further, given the camera matrix, the object/region bounding box and the depth map can generate object/region point clouds, which are sent to the plane fitter to produce ground-free equivalents. These ground-free point clouds can be used for:

1. pose estimation via template matching (future work)
2. dimension estimation to add as collision objects in manipulation server's collision scene
3. inputs for 6-DoF grasping pose inference based on grasp pose detection

The raw point cloud from sensors can

1. go through octomap directly, adding all to the collision scenes
2. go through plane fitter and octomap, adding only planes (ground, table, etc.) to the collision scenes

In addition to 3-DoF and 6-DoF grasping, the grasp server also supports 4,5-DoF grasping based on Dex-Net, which inputs the (bounding-box-cropped) RGB images and associated depth images.

Note that the grasp server only generates the grasp pose for the object. To plan and execute the grasp, we select the desired gripper grasp type (pinch, scissor, etc.), translate the pose to account for gripper-end-effector offset, and send it to the grasp executor, an example of the implemented online trajectory and action planning service. The object status topics are then used to check the success of the grasp.

After grasping the object, Amiga can perform other actions such as placing, stirring, pouring and more. Finally, it can navigate back to the master, finishing the entire task cycle.

⁵The 1D orientation can be directly computed

⁶The 3D orientation needs to be computed in an alternative way

5 Amiga Simulation

5.1 Background and Module Design

A *robotics simulator* is a digital tool used to engineer robotics-based automated production systems that operate without the need for a physical machine. Robotics simulations reduce the overall cost and time of integration; these applications can, in some cases, be transferred directly to physical robots without modification. 3D modelling and rendering of robots and their environments are among the most popular uses of robotics simulators. Software simulators of this type can spawn virtual robots capable of emulating the actual robots' motion in a natural work envelope. Several robotics simulators incorporate a physics engine to provide a more realistic movement of the robots. Whether an actual robot is available or not, developing automated control programs on a robotics simulator is usually recommended since it allows robotics programs to be conveniently written and debugged offline whilst the final program can be tested on an actual robot.

5.1.1 Comparing Simulators

To the best of our knowledge, there are six major robotics software simulators in the literature: Gazebo, RoboDK, Webots, CoppeliaSim, OpenRave and Unity. Clearly, learning a software simulator entails a significant opportunity cost, and the ideal scenario would be to obtain some standard of evaluation based on our requirements in advance.

Regarding this project, five success criteria for a software simulator are determined as below (ordering unimportant):

1. **Linux Support:** all computing devices in Amiga run Linux
2. **C++/Python API:** the rest of the modules are in these two languages
3. **Physics Engine:** to reflect and approach as close to reality as possible
4. **URDF File Import:** manufacturers have provided these files unitedly
5. **ROS Integration:** this project employs ROS as the middleware

All simulators support Linux, so this criterion does not filter out any candidate. In terms of programming languages and API, all of the above provide interfaces via C++ or Python except for Unity, which uses C# and hence is not selected. RoboDK does not provide ROS support nor a physics engine and is not selected. Further, Webots and OpenRave are not selected due to the lack of support for URDF files, and even though there is a way to convert

URDF files to their supported formats, importing nested URDF files may be problematic. The above analysis leaves us with two simulators, Gazebo and CoppeliaSim. As a matter of fact, a working implementation is established following the use of both simulators.

Gazebo (Figure 5.1) is an open-source 3D robotics simulator that integrated the ODE physics engine [ode, 2004], OpenGL rendering [opengl, 1992], and support code for sensor simulation and actuator control [Gazebo, 2022]. It utilises SDF (Simulation Description Format) as an extension to URDF to introduce additional simulation-based physical properties such as gravity, contact and friction. CoppeliaSim (Figure 5.2), on the other hand, is supported by an integrated development platform and features a distributed control architecture that allows individual models to be controlled individually by embedded scripts, plugins, ROS nodes, API clients, or custom-written solutions [CoppeliaSim, 2022]. Moreover, it also provides a direct programming interface using the Lua language.



FIGURE 5.1:
Gazebo



FIGURE 5.2:
CoppeliaSim

Remark

Both simulators use the ODE/Bullet physics engine and offer an intuitive scene editing tool for manipulating objects. However, one of the main disadvantages of CoppeliaSim is that it is very difficult to construct a custom camera plugin that publishes ROS simulated data topics such as RGB images and point clouds, which is, by contrast, much easier to accomplish in Gazebo via plugins specified in the URDF. Further, in CoppeliaSim, all scenes are saved in a unique V-REP format, so any scene editing must be achieved via the V-REP interface, which is inefficient for the development process. Due to these two factors, CoppeliaSim is ultimately given up.

In light of the above analysis, the final simulation component will be implemented using Gazebo, which satisfies all our needs (Linux, C++/Python API, physics engine, URDF, ROS support) and, most importantly, relatively easier to develop plugins for the sensors used in Amiga (directly on URDF).

5.1.2 Simulating Manipulation

Simulating arm manipulation is tricky. Compared to the mechanics of a walking robot that is limited to contact with the ground and a certain number of points, the mechanics of contact in manipulating objects are much more challenging to simulate. Even more significant was the centrality of perception in manipulating objects; it was generally agreed that the simulation of a camera was insufficient to yield meaningful results [Tedrake, 2021].

However, Robotics and computer vision communities have seen rapid adoption of video-game quality rendering in recent years. Now, it seems

that game engine renderers may have the capability to model cameras accurately enough not only to test perception systems in a virtual environment but also to train perception systems in a virtual environment and expect them to work in the real world and even achieve better performance than real-life data [Borkman et al., 2021].

Additionally, according to Professor Russ Tedrake [Tedrake, 2021], the quality and performance of contact simulation have also improved dramatically over the years. Simulating multibody contact with high precision and performance requires dealing with complex geometry queries and stiff differential equations. Although the mathematical formulations and numerical solutions for the governing equations still need to be refined fundamentally, solvers today are already good enough to be of great use.

Remark

For this particular component, the manipulation skills needed for simulation include robotic grasping, which is one of the more complex forms of object manipulation because it involves numerous contact points and a wide variety of geometry. Although, as pointed out above, the physics engine has been optimised for contact simulation, fine-tuning the robot URDF and objects' geometrical properties would also be necessary in order to perform a successful grasp. Tuning accurate physic-property parameters to achieve successful and realistic grasping is not the priority of this component. Therefore, we will need a solution that will eliminate this dependency on fine-tuning the URDF, so we may focus instead on examining the effectiveness of the motion planning component in achieving the desired grasp poses.

5.2 Implementation

5.2.1 Amiga Robot Model

A URDF file must be created in order to allow Amiga to spawn in a software simulator. Amiga's arm, gripper and wheelbase manufacturers have provided the URDF files associated with their hardware. However, two additional tasks need to be accomplished. First, we must create a URDF for the custom components, such as the 3D-printed camera mount; second, we must merge all the separate files into a single URDF.

Developing URDF: To start, let us develop the URDF files for the camera mounts. As Preliminary 2.1 discussed, the mounts can be viewed as links composed of inertial, visual, and collision components. The visual and collision information can be provided by a DAE or a STL file, which is natively generated by the 3D CAD software specialising in *stereolithography*¹. Most CAD software interfaces smoothly with the 3D printers; therefore, the files originally created for printing the camera mounts could be converted into

¹A 3D printing technique that creates models, prototypes, patterns, and production parts in a layer-by-layer manner

ROS-compatible formats such as DAE and STL². Moreover, we employed MeshLab [MeshLab, 2022], an open-source system for processing and editing 3D triangular meshes, to edit and render the meshes. The surface geometries contained in the resulting STL files are visualised as in Figure 5.3. These files can be integrated into a URDF through the `<geometry>` tag.

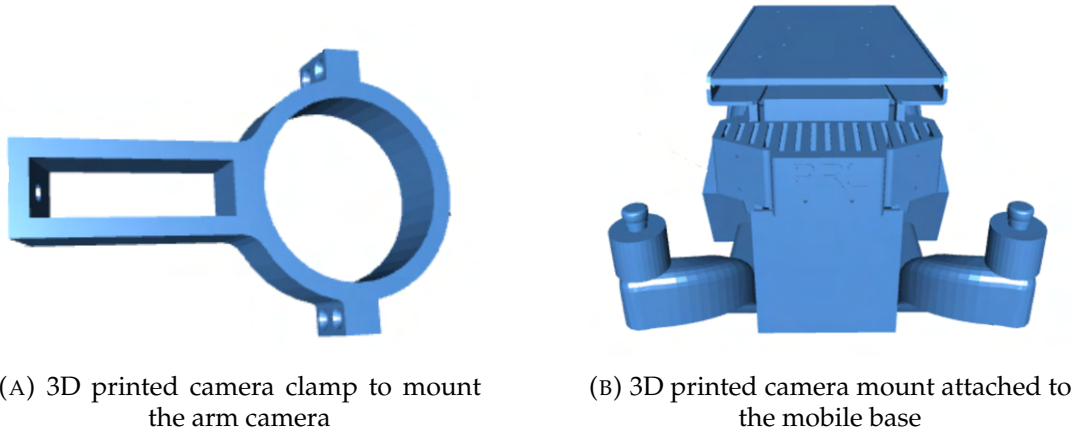


FIGURE 5.3: Visualisation of the STL files for Amiga's custom components

After taking care of the visual and collision aspects, we need to provide information on inertia. Using inertia formulas derived from commonly found shapes [Inertia, 2022] and the fact that the inertia for a composite shape is equal to the sum of the inertia for its constituent parts, we could estimate the moment of inertia for the above objects. Moreover, we will need to account for the hollow shapes by subtracting the holes inside.

Many more URDF files are created following the above processes.

Combining URDF: There are dozens of URDF files, each of which describes a different hardware component of Amiga. It seems desirable that all of them could be gathered in one document. The solution is to use xacro [xacro, 2022], an XML macro language that allows the construction of shorter and more readable XML files by using macros that expand to more significant XML expressions. We organised and converted all the URDF files to xacro files in a hierarchical manner and used prefixes to avoid ambiguity in naming. In this regard, the work is quite time-consuming since URDF files from different manufacturers may be based on different standards, and some may have their own issues. As an example, for the UR10e arm, there exists a joint limited version that uses joint limits restricted to $[-\pi, \pi]$ because the full joint version fails to handle motion planning.

Once the all the sub-xacro files had been completed, we could import each one into a high-level xacro file. For example, for the camera clamp:

```
<xacro:include filename="path://camera_clamp.urdf.xacro" />
```

²DAE files are basically more complex formats than STL files and contain much richer information such as colour and texture

Additionally, since several of the xacro files refer to a link, we must declare a joint to connect that link with another. For instance, the camera clamp should be attached to the arm as follows:

```
<joint name="arm-clamp-joint" type="fixed">
  <origin xyz="0 0.15 0" rpy="0 -2.35619 3.14159" />
  <parent link="amiga_arm_wrist_3_link" />
  <child link="amiga_grip_camera_clamp_link" />
</joint>
```

The location and rotation of the simulated robot must be adjusted several times to make it resemble the actual one. The same procedure was repeated for all other joints required.

The above development was quite labour-intensive, but it paid off; as seen in Figure 5.4 below, it is now possible to visualise Amiga's robot model³:

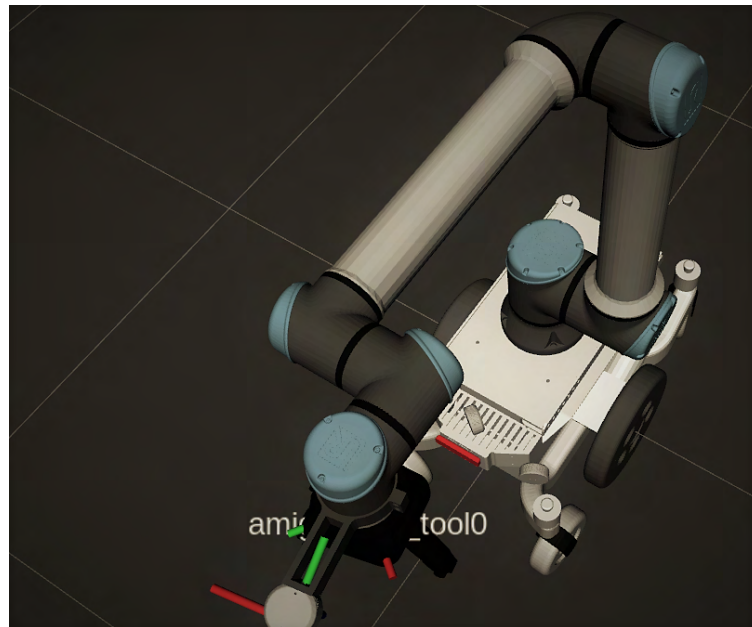


FIGURE 5.4: Rviz visualisation of Amiga's robot model. Only one URDF file is uploaded to the ROS parameter server

5.2.2 Simulation in Gazebo

Recall in Background 5.1.1 that Gazebo was selected as the robotic simulator following a series of evaluations. One of the most significant benefits of Gazebo is that it allows directly modifying the URDF files to enable Gazebo-specific features via a simple `<gazebo>` tag. When the project is launched in the simulation environment, we can set an arg parameter in the xacro file to true. This makes sense as we do not want to run the simulated versions when real hardware is connected, and switching between the two is straightforward:

³Note the two l515 cameras' transforms have not been calibrated yet

```
<xacro:arg name="gazebo" value="false" doc="true for simulation"/>
```

Adding Virtual Actuators The URDF file from Amiga can now be directly loaded into Gazebo. However, the robot will be crumbling within the simulated world. For this issue to be addressed and Amiga to be controlled in Gazebo, we require the virtual motors to be operational. This can be achieved by the `ros_control` plugin [ROS_Control, 2013]:

```
<plugin name="gazebo_ros_control" file="libgazebo_ros_control.so">
```

For each joint we want Gazebo to act upon, we need to add a `<transmission>` block similar to the following:

```
<transmission name="trans_amiga_gripper_finger_1_joint_1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="amiga_gripper_finger_1_joint_1">
    <hardwareInterface>hardware_interface/
      PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="amiga_gripper_finger_1_joint_1_motor">
    <hardwareInterface>hardware_interface/
      PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

The above lines add `PositionJointInterface` for commanding one of the position-based joints within the gripper group. We add `<transmission>` to all the joints we wish to control. We use `PositionJointInterface` for the joints on the UR10e and gripper group and `EffortJointInterface` for the wheel joints.

Fine-tuning Dynamics Now that Amiga can be spawned statically, some dynamic parameters must be set in the proper way to simulate motion. This is probably the most challenging aspect of the simulation module - there were no official tutorials, and all we had was the Gazebo community for inspiration. Following are the two most significant difficulties encountered and the solutions devised:

- **Wheel Immobility:** This was a common problem, and the reason was either that the wheels were not flat or the friction and torque were set in an incorrect manner. First, we should ensure that the wheels were in contact with the ground. If they turned out to be hanging above the ground, we should manually correct the contacting joints and positions in the URDF. In addition, we ought to review the friction and torque parameters for the wheel links. Even though Amiga only has two motor wheels, all six wheels' dynamics must be set to enable movement in order to enable movement. We add the following lines to all the wheel

links, where μ_1 and μ_2 are friction coefficients for the principal contact directions along the contact surface, minDepth is the minimum allowable depth before contact correction impulse is applied, k_p is the contact stiffness, and k_d is the damping for rigid body contacts:

```
<gazebo reference="${prefix}drive_wheel_left">
  <mu1>1000.0</mu1>
  <mu2>1000.0</mu2>
  <minDepth>0.003</minDepth>
  <kp>100000000.0</kp>
  <kd>1</kd>
</gazebo>
```

- **Gripper Shaking:** the robotiq 3F gripper vibrated violently in the simulation world. The reason for this was that the physics parameters were incorrect and certain joints had self-colliding effects. The repository provided by robotiq [robotiq_github, 2019] was no longer maintained, particularly for its most recent product, the 3F gripper. Several users experienced the same difficulties without any resolution. We solved this issue by increasing `<damping>` in the original URDF file and disabling self-collision of links as follows:

```
<gazebo reference="amiga_gripper_palm_finger_1_link_0">
  <self_collide>0</self_collide>
</gazebo>
<gazebo reference="amiga_gripper_palm_finger_2_link_0">
  <self_collide>0</self_collide>
</gazebo>
<gazebo reference="amiga_gripper_palm_finger_middle_link_1">
  <self_collide>0</self_collide>
</gazebo>
<gazebo reference="amiga_gripper_palm">
  <self_collide>0</self_collide>
</gazebo>
```

The simulation functions on Amiga were basically all set following the above fixes. The next step was to spawn some objects. Despite Gazebo's availability of many models, few kitchen-scene objects were included. We could create our models by writing URDF files. As a result, we built a banana and a coke model using Blender [Blender, 2021], which allows for rendering images and mesh files into Gazebo-compatible DAE files.

Implementing some simple controls, such as the keyboard teleops, would be beneficial. In the lab, teleops for moving the wheels had already been implemented, so as part of our contribution, we also created the arm teleop. Moreover, we also built various simulation playgrounds for testing other modules. The teleops and developments of the Gazebo simulation environments throughout the project are described in Appendix A.1.1.

5.2.3 Camera Plugin

Although our depth cameras have URDF files, they are merely object models without any functionality. Therefore, it is necessary to develop the camera sensor plugin to capture actual images. There is a Realsense D435i plugin from Intel [D435, 2020], but not one for the L515, so we will need to modify it to meet our requirements.

This process entailed that we set parameters such as the camera resolutions, encoding formats, and reference frames, among others. Afterwards, we enable it by compiling the binary files into a .so file and writing something like the following to the URDF:

```
<gazebo>
  <plugin name="l515_grip" file="librealsense_gazebo_plugin.so">
    <prefix>"l515_grip"</prefix>
    <depthImageParameter>...</depthImageParameter>
    <colorImageParameter>...</colorImageParameter>
    <infraredImageParameter>...</infraredImageParameter>
    <pointCloudParameter>...</pointCloudParameter>
    . . . . .
  </plugin>
</gazebo>
```



FIGURE 5.5: Amiga in Gazebo world with camera plugin. The point clouds of the simulation world is captured and published as ROS message topics

Moreover, there were two optional things to do: camera intrinsic and hand-eye calibration. As for intrinsic calibration, we simulated a calibration grid and used the ROS camera calibration package [ROS_Calib, 2020]; the results were then saved to a YAML file and broadcast to ROS via the camera_info topics as specified in the plugin. For hand-eye calibration, we used the aruco_ros package [aruco_ros, 2021] and easy_handeye package [easy_handeye, 2021], which enabled Amiga to track a moving marker whose pose is known at any moment. Since these were merely nice-to-haves, we will not delve too deeply into them. Figures are provided in Appendix A.1.2

for each process. The calibrations have resulted in a more accurate camera in simulation.

Finally, we can apply this plugin to the camera link via the `<gazebo reference="l515_camera_link">` tag. Figure 5.5 above demonstrates Amiga spawned in the Gazebo simulated kitchen scene where point clouds of scene objects are recorded and published in ROS and displayed in Rviz.

5.2.4 MoveIt Integration

So far, we have only been able to control Amiga's arm using the keyboard teleop, which is of limited use. In order to test the Amiga manipulation module in the simulation, we need to set up the communication between Gazebo and MoveIt⁴, which is the main dependency of the motion planning module.

To achieve this, we must write three necessary controller configuration files: Follow Joint Trajectory, Joint Trajectory Controller and Joint State Controller. These files are available in appendix A.1.3. We use position controllers for the arm and effort controllers for the gripper.

The three controllers function as follows:

- **Follow joint trajectory:** a client of action communication and responsible for sending MoveIt's planned trajectories to Gazebo
- **Joint trajectory controller:** the server of the action communication and is responsible for loading the configuration parameters and launching the controller
- **Joint state controller:** publishing the joint states of the robot at 50HZ

Figure 5.6 illustrates the communication in the arm's case. The motion planning results are sent to Gazebo, which broadcasts the most recent joint states. The joint state controller then publishes the transforms of joints.

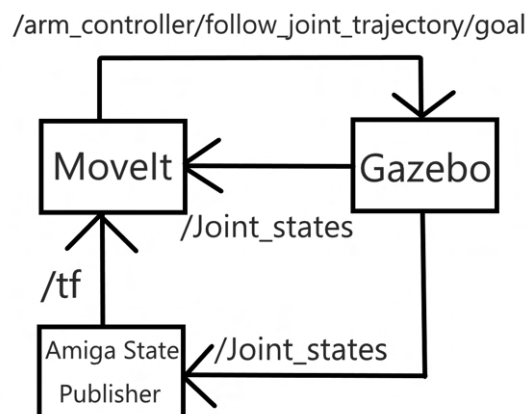


FIGURE 5.6: Illustration of our MoveIt - Gazebo integration

⁴We will shortly introduce MoveIt in the Amiga manipulation chapter

5.2.5 Grasp Plugin

Now that Gazebo has motion planning working, it may be possible to grasp an object. However, grasping was difficult in Gazebo. In the event of contact, the objects would wobble, explode, or fly away into space.

Recall in Background 5.1.2 we argued that fine-tuning Amiga's URDF and the objects' geometrical properties would be necessary in order to simulate a successful grasp, but this is not the priority of the simulation component.

Fortunately, several developers also observed this issue and established repositories to facilitate the resolution. One option was to use Grasp_Gazebo [Grasp_Gazebo, 2018], where the DART physics engine is integrated with Gazebo to offer tools for dynamic grasp simulation. However, this package was designed to simulate advanced realistic grasping techniques, which were beyond our requirements. The solution we finally employed was to use the Gazebo grasp fix plugin [Grasp_fix, 2021], which is designed to fix objects gripped by robot hands to avoid errors with physics engines as well as to help the objects stay in place in the hands of the robots.

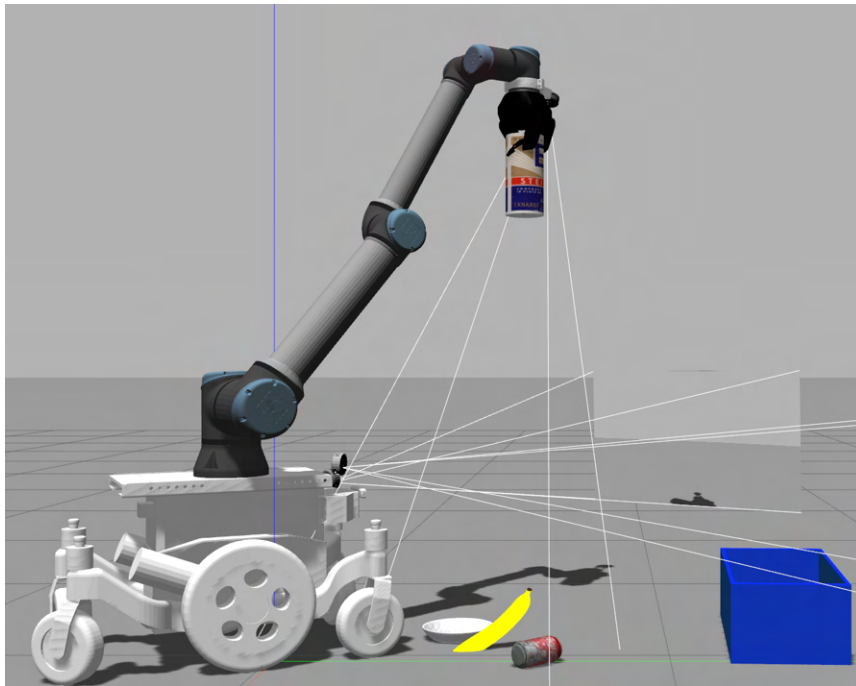


FIGURE 5.7: Demonstration of Amiga successfully grasping a beer in our Gazebo simulation environment

Once the gripper links apply two opposing forces to an object, that object is considered "grasped" and is fixed to the hand link. Once this criterion no longer holds (e.g. the gripper opens), the object is again detached. The opposing-force criterion is parameterised (angles, thresholds, tolerances, etc.) and configurable. Our configuration results were obtained after extensive testing. Figure 5.7 presents an example of successful grasping in simulation. With the help of the grasp plugin, we can examine the effectiveness of

the motion planning and grasping modules (since the objects will no longer bounce off immediately upon contact with the gripper).

5.2.6 LiDAR plugin

Similar to setting up the camera plugin as in Section 5.2.3, we also incorporated the plugins for LiDARs. Fortunately, the gazebo plugin for rplidar (recall Chapter 3) has already been developed in the [ROSbot_2.0, 2022] package. We therefore adopted it directly and integrated it into Amiga's URDF under the <Gazebo> tag (the transforms are adjusted visually). Figure 5.8 illustrates the LiDAR plugin results. The walls in the simulation environments are scanned by the LiDAR and published in ROS as sensor_msgs/LaserScan Message. The incorporation of this LiDAR plugin means it is possible to test the navigation modules in simulation.

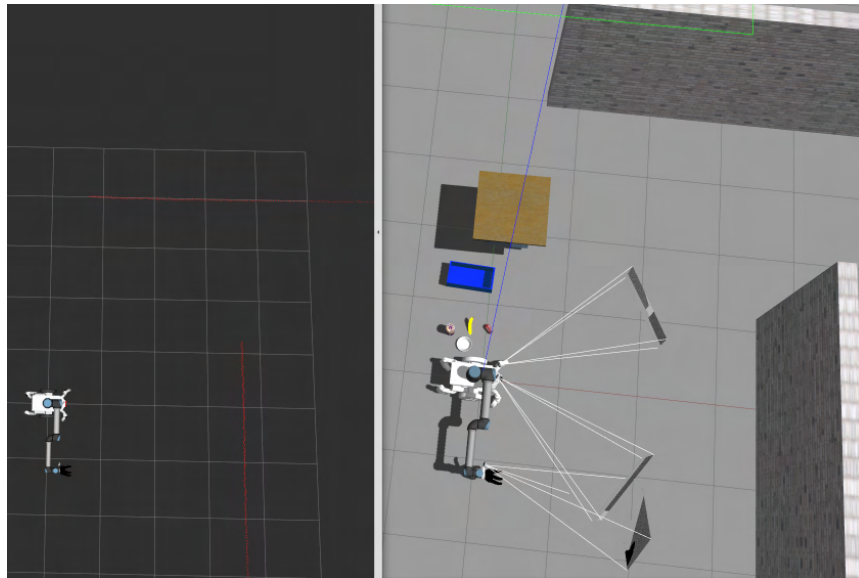


FIGURE 5.8: Amiga in Gazebo world with LiDAR plugin. The right-hand side is the scene and the left-hand side is the visualisation of the corresponding laser scan messages

5.3 Architecture Summary and Comments

To conclude, we have implemented a simulation module in which Amiga can perform various tasks in a virtual kitchen environment. Figure 5.9 illustrates how this module is connected to other modules inside our final system as introduced in Section 4.2, which are designed to operate on the real robot.

As shown in the figure, switching between the real robot and the simulation environment can easily be achieved by setting a parameter `sim:=true/false` when loading Amiga's URDF. Moreover, the connection of the simulation module to other modules (1) - (4) are summarised as below:

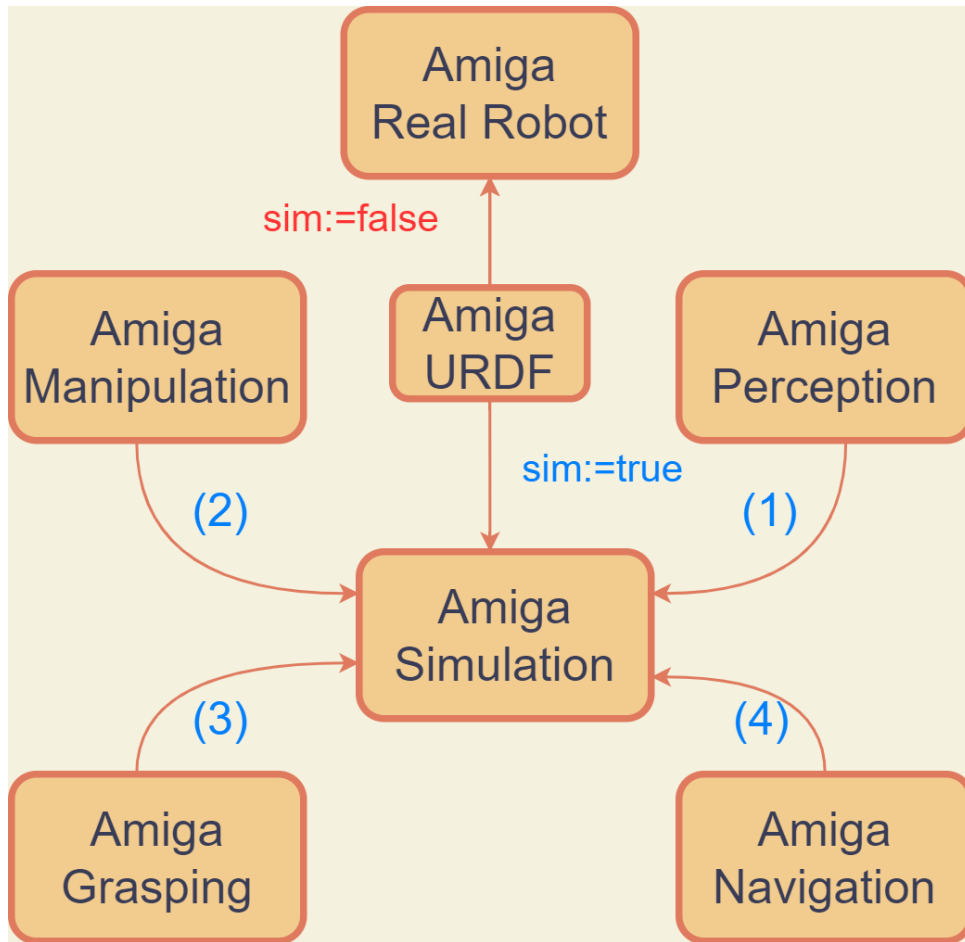


FIGURE 5.9: Connections between Amiga simulations and other modules. Tests and debugging of all other modules in our system can be performed offline in the simulation environment. Switching between the real and the simulated robot is achieved by a single parameter as specified in the URDF

- **(1) Perception - simulation link**

Established: created the L515 camera plugin, whose accuracy has been improved by performing intrinsic and hand-eye calibrations in the simulation environment. Since Amiga perception does not depend on the type of camera used, full integration is possible. To do: implement the ZED2 camera plugin and account for uncertainties in real sensors

- **(2) Manipulation - simulation link**

Established: established Gazebo-MoveIt communication, implemented the virtual motors and expanded the SRDF for simulation's specific purposes. In addition to simple keyboard teleop, the simulation module allows complete Amiga manipulation support. The only difference is that in simulation, the gripper is positionally controlled through MoveIt, whereas in the real world, we communicated to the hardware directly (to allow more flexible control as characterised by the Robotiq 3f gripper such as different modes, forces and velocities)

- **(3) Grasping - simulation link**

Established: fully supported our integrated vision-based grasp pose inference algorithms such as Dex-Net and Grasp Pose Detection. Near-realistic grasps were achieved via the grasp plugin. To do: build more dexterous, realistic and dynamic grasping as in [Grasp_Gazebo, 2018]

- **(4) Navigation - simulation link**

Established: fully supported the navigation module in addition to keyboard teleop, although ZED 2 camera's inherited visual odometry is not supported To do: build an accurate map for the simulation kitchen (the process can be difficult for now due to low FPS in the large world)

Initially, the Amiga simulation module was developed because the real robot would not be available until February 2022. However, it turned out to be extremely useful. As the project progressed, this platform was extensively utilised for prototyping and debugging different algorithms, which greatly facilitated the development of the real robot side as well. This will become more apparent in the next few chapters, which introduce other components. Furthermore, due to the imperfection of the simulation environment, a Sim2Real⁵ gap is unavoidable. This will be discussed in more detail in the evaluation chapter.

⁵A concept of transferring robot skills acquired in simulation to the real robotic system

6 Amiga Perception

6.1 Background and Module Design

Computer vision is an interdisciplinary scientific field that deals with how computers can gain a high-level understanding from digital images or videos [Ballard and Brown, 1982]. A high-level understanding may take the form of either a simple signal (good/bad regions) or a more complex set of data such as the identity (*image classification*), bounding-box level identity (*object detection*) and even pixel-level identity (*image segmentation*) of each object in the image. Furthermore, in a robotics context, and as it applies to this project, it would also be of great interest for us to determine from a (kitchen) scene an object's real-world 3D position and 3D orientation (*pose estimation*) in order to perform manipulation tasks without possible collisions.

6.1.1 Object Detection

Object detection classifies objects from a scene image and puts 2D bounding boxes around them. Following Preliminary 2.3, the pixels in the boxes can then be mapped to spatial coordinates of the entities of interest.

Several object detection algorithms are available today, including traditional image processing and deep learning methods. A traditional object detection pipeline consists of three phases: selection of informative regions, feature extraction, and classification [Zhao et al., 2018]. First, a multi-scale sliding window is applied to scan the whole image. Then, representative visual features are extracted using feature descriptors such as SIFT [Lowe, 2004], HOG [Dalal and Triggs, 2005] and Haar [Viola and Jones, 2001] and finally a classifier like SVM [Cristianini and Ricci, 2008] and AdaBoost [Schapire, 2013] is used to distinguish amongst possible categories.

Traditional detection methods have proven successful in the industry, where fast, precise and steady algorithms are favoured. However, one of the lethal limitations is that the manually engineered low-level descriptors struggle to generalise outside a fixed biased industrial environment where objects come in a wider variety of appearances, illuminations, and backgrounds. This issue has been resolved by turning into deep learning methods, where many layers of CNNs are applied to extract features on an automatic basis. Researchers soon realised that CNNs works particularly well for images as local neighbourhoods are highly correlated and proposed a variety of learning-based object detection algorithms.

Contemporary deep learning-based object detection algorithms can be divided into two subcategories: region proposal and regression/classification

methods. Region proposal-based methods are also known as two-stage detectors and follow a traditional object detection pipeline by first generating a region proposal from a number of pre-set anchor boxes and then classifying it into different categories. Common examples include the RCNN [Girshick et al., 2013], Fast RCNN [Girshick, 2015], Faster RCNN [Ren et al., 2015], Mask RCNN [He et al., 2017], SPP-net [He et al., 2014] and FPN [Lin et al., 2016]. On the other hand, regression/classification methods are referred to as one-stage detectors and output categories and locations directly via a single unified framework. YOLOv2 [Redmon et al., 2015], YOLO9000 [Redmon and Farhadi, 2016], YOLOv3 [Redmon and Farhadi, 2018], YOLOv4 [Bochkovskiy, Wang, and Liao, 2020], YOLOV5 [Zhu et al., 2021], YOLOX [Ge et al., 2021], and SSD [Liu et al., 2016] are examples.

As it seems, plenty of models are available on the market. Specifically, with respect to this project, four success criteria for an object detection model are determined as below (ordering unimportant):

1. **Latency Performance:** must run fast on available computing devices
2. **Pre-trained classes:** if using pre-trained models, classes contained should incorporate as many common kitchen scene objects as possible
3. **Training and Deployment:** When training a customised Dataset, make use of lightweight backbones and deployment-friendly architectures
4. **ROS Integration:** preferably already integrated into mature ROS wrappers to avoid reinventing the wheels

Oftentimes, there exists a trade-off between model accuracy and latency. This trade-off was evaluated thoroughly by Google research in 2016 in a paper entitled "*the speed/accuracy trade-offs for modern convolutional object detectors*" [Huang et al., 2016]. A list of key findings from the paper is summarised below to assist our model selection decision-making:

- **Overview:** Due to the existence of region proposal networks, two-stage detectors are generally slower than one-stage counterparts; e.g. Faster R-CNN requires at least 100ms per image.
- **Most Accurate:** With 300 proposals, Faster R-CNN using Inception ResNet is the most accurate single model. A single image takes approximately 1 second to process.
- **Fastest:** SSD with MobileNet provides the best accuracy trade-off within the fastest detectors.

Accordingly, two-stage detectors are unsuitable as they have much higher latency than one-stage detectors. Moreover, amongst the candidate one-stage detectors, YOLO and SSD have very mature ROS wrappers [yoloROS, 2021][detectNet, 2019] deployed in CPU/GPU and Jetson on-board computers (Section 3.1.4). Both repositories provide pre-trained models on the COCO

Dataset [Lin et al., 2014], which contains 80 classes of commonly seen objects. In case the 80 classes involved are not as relevant to our kitchen tasks, possibilities of creating a custom Dataset are also explored, which include the Open Images Dataset [Kuznetsova et al., 2020] [Krasin et al., 2017] which contains more than 15,851,536 boxes on 600 categories, and EPIC-KITCHENS Dataset [Damen et al., 2021b][Damen et al., 2021a] which contains a large-scale data which includes 97 verb classes and 300 noun classes in egocentric vision.

Remark

For this project, models' performances (both in accuracy and speed) in detecting real-life kitchen objects remain unknown until the integrated system is tested with a live camera stream. The unique eye-in-hand mounting configuration of the camera implies that the camera's viewpoints differ considerably from traditional static images. Therefore, even the most state-of-the-art models may perform poorly upon testing. As much as possible, it should be avoided that models are later discovered to be unsatisfactory after a long deployment and integration process. Consequently, a good strategy for developing the object detection modules would be to construct several quick prototypes, test them, and then refine them if necessary. The YOLO and SSD repositories mentioned above are excellent places to start.

As an additional point, it is worth clarifying that the above discussions assume object detection is performed in 2D and that images are the input. As a matter of fact, there exist 3D detection methods that utilize point clouds as inputs [Lang et al., 2018] [Shi, Wang, and Li, 2018]. To a greater extent, there are also fusion methods which consume both images and point clouds as input [Liang et al., 2019][Qi et al., 2017]

We have omitted these methods for two main reasons: first, the models are difficult to deploy with the existing hardware settings and would probably run very slowly; and second, if the models do not perform well, it is much more difficult to find 3D kitchen-scene point cloud Dataset to refine the models than its 2D image Dataset counterpart.

Additionally, monocular 3D object detection methods based on RGB images [Bao, Yu, and Kong, 2020][Qin, Wang, and Lu, 2019] are not considered for this project because it is unnecessary to estimate depth information from RGB images, which is already provided accurately by the RGB-D cameras.

6.1.2 Pose Estimation

Having managed to produce the 3D position, one may also desire the 3D orientation. The combination of a 3D position and a 3D orientation is referred to as the 6D pose of an object, with which the object can be transformed from the object coordinate into the camera coordinate. A robot may utilize this information to manipulate an object so that it does not interfere with the object.

6D object pose estimation is instrumental in robotic and augmented reality fields. According to [Du et al., 2020], 6D object pose estimation can be divided into three kinds: correspondence-based, template-based and voting-based methods.

The correspondence-based methods entail finding correspondences between observed input data and the complete 3D model of the object and recovering the 6D object pose through Perspective-n-Point (PnP) algorithms [Lepetit, Moreno-Noguer, and Fua, 2009]. It is particularly effective when the target object possesses rich textures or geometric details. By contrast, template-based methods are helpful when the target object lacks detailed texture or geometry. In contrast, template-based methods, which identify the most similar template from the templates labelled with ground truth 6D object poses, are useful when the target object lacks detailed texture or geometry. In voting-based methods, each pixel or 3D point contributes one or more votes to estimating the 6D pose of the object, which works well in cases where the object ranges between specific objects and category-level objects.

Of course, all of the above methods entail traditional and deep learning approaches. As an example, DOPE [Tremblay et al., 2018] is a deep learning template-based method for 6-DoF pose estimation of known objects from an RGB camera. It utilises simulated environments to generate virtual data, including racker boxes, sugar boxes, tomato soup cans, mustard bottles, etc.

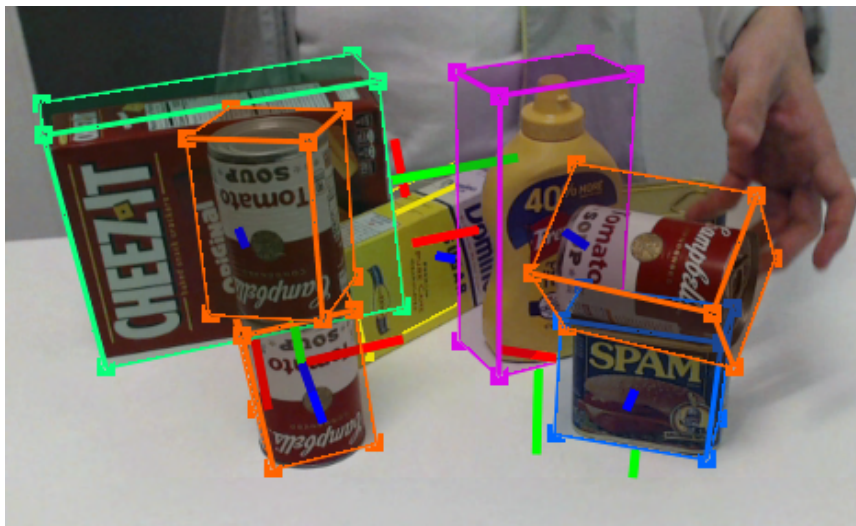


FIGURE 6.1: Deep Object Pose Estimation (DOPE) example output [Tremblay et al., 2018]

Three use cases of pose estimation in this project have been come up:

1. To be used as the visual basis for collision avoidance module
2. To be used as the visual basis for robotics grasping module
3. Used for checking whether the object has been manipulated into the desired pose after a manipulation task

Remark

However, a significant concern in applying pose estimation to this project is the lack of a unified approach to combining object detection and pose estimation in one component module. An object detection model that only works on bananas and a

pose estimation model that only works on bottles will not construct a good use case. One strategy could be building a CAD model databases for all objects that can be confidently detected, and employing template-based methods. However, this is based on a mature kitchen-scene object detection model, which we don't have yet.

As a result, for the component design of this project, alternatives of 1. and 2. are founded and introduced in Section 7.2.9 and Section 8.1.1, respectively. A unified computer vision module that detects and estimates poses for a list of known objects will remain open as an exciting direction for the future development of this project.

6.2 Implementation

6.2.1 Camera Configuration

The first thing we want to do is to set up the camera correctly. Recall that we mainly used two cameras, Realsense L515 and ZED 2.

We install the L515 and ZED 2 dependencies and clone their ROS wrappers [Realsense_ROS, 2018] [ZED_ROS, 2022]. Since multiple cameras are used, the launch files are different depending on where the camera is connected. Hence, two versions are developed and dockerised for each Jetson used for manipulation and navigation.

Moreover, the L515 camera, by default, publishes unaligned depth images and unordered point cloud messages. We need to set the `<align_depth>` and `<ordered_pc>` arguments to true. On the other hand, the ZED 2 camera already publishes these messages. The topics that were used the most are listed in Appendix A.2.1. It should be recalled that the camera on the arm was changed from l515 to ZED 2 because it overheated and disconnected frequently. All the vision-based algorithms support both cameras by including a configurable argument parameter `<camera>`¹.

ZED 2 cameras use a different coordinate system from ROS; this caused confusion, for which we had to develop another set of data-processing codes for ZED 2's use only. However, it was later discovered from the source code that we could add the following line and re-compile it such that the ZED 2 will use the ROS standard instead:

```
init_params.coordinate_system=COORDINATE_SYSTEM::RIGHT_HANDED_Z_UP;
```

We have suggested the manufacturers to augment the launch file with the capability of switching between different coordinate systems.

Also of note is ZED 2's neural depth sensing mode, which utilises a lightweight neural network to reproduce human vision. After some optimisations in the camera's arguments, such as `<max_depth>`, we were able to run this mode smoothly. Compared to its other modes and L515, the neural mode produces better quality depth images with more details and fewer noises.

Moreover, the ZED 2 camera offers positional tracking functionality, which publishes the 6DoF pose of the camera and hence could be used as visual odometry for navigation. Compared to the original Realsense T435 cameras planned for navigation use, it is more accurate and stable.

¹Set it to "ZED 2" makes all the modules subscribe to the ZED 2 camera topics

6.2.2 3D Coordinate Mapping

By following rigorous camera calibration procedures based upon industry standards, manufacturers of these RGB-D cameras already provided very accurate camera parameters, including distortion coefficients. In turn, by applying the camera model (Preliminary 2.3), accounting for distortion, and utilising the depth map (Preliminary 2.4) collected, accurate bridging between 2D pixels and 3D spatial coordinates can be established. This concept builds the core foundation for the perception module.

Suppose now we have a RGB image $f(x, y)$, an aligned depth image $g(x, y)$, a point cloud message $pc(x, y, z)$ and a 2D pixel location, u and v . We could first retrieve the depth by $d = g(u, v)$. Given u, v, d , we devised two methods to compute the 3D coordinate xyz . The first method is based on point cloud messages published from the camera, bypassing the depth images. This method is shown in Appendix A.2.2 and is tested to be less stable (returning a lot of NaNs) and five times slower than the other approach, which utilises the depth maps. This underperformance was because the point cloud message was itself generated using the RGB and depth images; hence, the errors and latencies during generation would carry on to the reading stage.

Depth Map Approach: this method uses the camera_info/depth image messages, and depends on the image_geometry library [image_geometry, 2009]:

```
import PinholeCameraModel

model = PinholeCameraModel()
model.fromCameraInfo(cam_info)

def uvd2xyz(u, v, d)
    ray = model.projectPixelTo3dRay(u, v)
    # normalize the ray so its Z-component equals 1.0
    ray_normalised = [i / ray[2] for i in ray]
    # multiply the ray by the depth; its Z-component
    # should now equal the depth value
    xyz= [i * d for i in ray_normalised]
    return xyz
```

Nonetheless, the depth map may sometimes return NaN values; this can be addressed by using a smaller window and reading the neighbourhood values. Stability can be improved by reading multiple times and averaging.

6.2.3 Kitchen Object Detection

Recall that in Section 6.1.1 we proposed an adequate strategy for developing the object detection modules, which includes constructing several quick prototypes, testing them, and then refining them if necessary.

Attempts were made to ZED 2 SDK [ZED2_ObjectDetect, 2022], YOLO v3/YOLO 9000 ROS wrappers[yoloROS, 2021] and DetectNet [Nvidia, 2019].

ZED 2 camera SDK provides a 3D Object Detection API, but currently, it can only detect and track a limited number of objects and is not very useful here. YOLO v3 and YOLO 9000 were implemented in the same repository, and switching between the two was straightforward by modifying and downloading a different config and weight file. Launching this node and subscribing to our cameras' image topics yield Figure 6.2, which illustrates the results applied in the simulation environment.

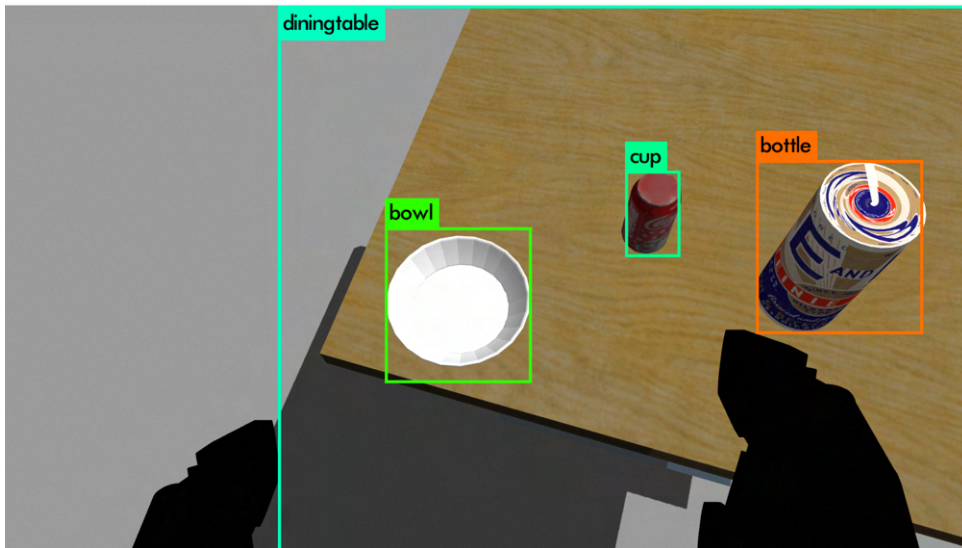


FIGURE 6.2: YOLO v3 object detection applied in Amiga simulation. Detected objects belong to the COCO classes

The next step was to download CUDA [CUDA, 2022] on our user laptop, which runs a dual-booting system and has a RTX 3070 graphics card. After that, we modified the CMakeLists.txt and added the following line:

```
-O3 -gencode arch=compute_81,code=sm_81
```

where the number 81 is the capability version of our GPU. Recompiling and launching again improved YOLO object detection's fps from 0.1 to 50.

The DetectNet is an object detection model provided by Nvidia for its Jetson devices. The integration to ROS was achieved by [ros_deep_learning, 2022], which depends on the vision_msgs. We sshed to the Jetson responsible for the arm camera and configured the necessary DetectNet setup with our cameras. Subsequently, a number of experiments were conducted to test the models' performance with a top-down view from Amiga. The initial evaluations of these models with real world objects are summarised in Table 6.1:

The YOLO V3 model performs the best amongst the three candidates in terms of accuracy²; tests of the YOLO 9000 model, which contains 9000 classes, revealed too many false positives, and the DetectNet from Nvidia proved ineffective in detecting anything except teddy bears and persons.

²Full evaluation will be discussed in the evaluation chapter

Models	Hardware Used	Speed	Performance Capture
YOLO V3	RTX 3070	50 fps	Relatively acceptable
YOLO 9000	RTX 3070	20 fps	Many false detections
DetectNet	Jetson Xavier NX	150 fps	Not very detective

TABLE 6.1: A comparison of object detection models so far

Moreover, both YOLO V3 and DetectNet were pre-trained on the COCO dataset [Lin et al., 2014], which contains 80 classes of commonly-seen objects. As our context is in the kitchen scenes, only tiny portions of the classes are relevant, which are summarised in Appendix A.2.3. On the other hand, YOLO 9000 contains too many unnecessary classes and is not robust enough for our purposes. In response to these findings, we were prompted to train a model exclusively for use in the kitchen context.

KitchenNet: Figure 6.3 illustrates our training strategy of KitchenNet, a customary model we made for Amiga’s perception in kitchen scenes.

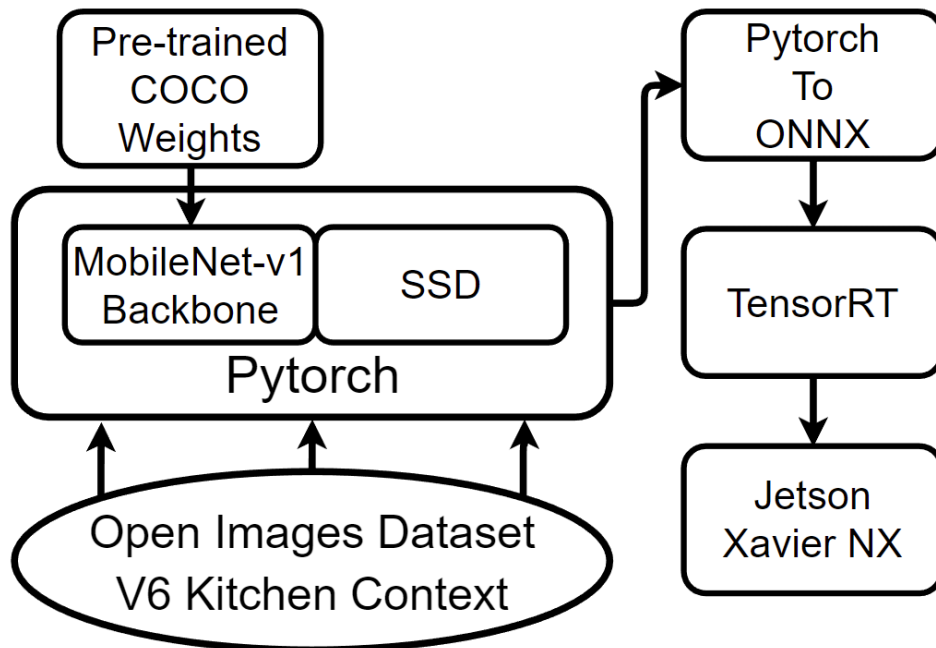


FIGURE 6.3: Training and deployment strategy for KitchenNet

Following the arguments in Preliminary 2.6 and background in 6.1.1 we chose a lightweight MobileNet backbone coupled with a SSD architecture. This combination was then implemented in Pytorch³[Paszke et al., 2019] and loaded with the pre-trained COCO weights.

³An open-source deep learning framework

Next, we prepared a kitchen-related object Dataset manually selected from the Open Images [Kuznetsova et al., 2020], which contained 600 object classes on 1.9M images. This kitchen object of interest Dataset includes 37 categories and 100K images, the distribution of each class in the training Dataset is shown in Appendix A.2.3. As a result of OpenImage being unbalanced, our Dataset was unbalanced as well, with 10 classes having fewer than 1k images and 15 classes having between 5k and 8k⁴ images.

To address this, common model training tricks such as *focal loss* and *data augmentation* were applied via Pytorch to reduce the uneven Dataset problem further. The model was then trained with batch size 32 for 100 epochs on a laptop with RTX 3070. The training took 75 hours, and the log was shown in Appendix A.2.3 as well.

This training model was saved as a .pt file as a result of using Pytorch. While it is possible to install Pytorch on Jetson and use it, inference will be slow. Instead, we converted the .pt file to .onnx⁵ and employed TensorRT [TensorRT, 2022] from Nvidia, which features post-training quantisation, kernel auto-tuning and tensor fusion, to optimise the model inference. The optimisation results were then saved into a cache file so that every time the model launches, it would automatically load.

Resultantly, the final model deployed on Jetson runs at more than 200 fps. Further details regarding the performance of this model will be discussed in the evaluation chapter.

Once the object detection model is online, we could subscribe to their serialised outputs (bounding boxes, detected objects, probability, etc). A code snippet would look something like this:

```
import rospy
from vision_msgs.msg import Detection2DArray
from darknet_ros_msgs.msg import BoundingBoxes

class ExampleNode:
    def __init__(self) -> None:
        rospy.Subscriber("darknet_ros/bounding_boxes",
                        BoundingBoxes, self.callback_darknet)
        rospy.Subscriber("obj_detect/kitchenNet/detections",
                        Detection2DArray, self.callback_detnet)

    def self.callback_darknet(self, data)
        # deserialise data according to message definition..
    def self.callback_kitchenNet(self, data)
        # deserialise data according to message definition ..
```

Lastly, it is worth mentioning that common object detection Dataset (including ours) is primarily a collection of static images. However, since Amiga is moving, its viewpoints may sometimes differ significantly from the Dataset.

⁴8k was our upper limit to reduce Dataset imbalance

⁵A format designed to represent any type of Machine Learning and Deep Learning model

The epic-kitchen [Damen et al., 2021b] is a large-scale Kitchen context dataset in first-person vision that matches our needs highly. One of the participating teams released their Fast RCNN-based models based on Tensorflow in [Object_Detection_Model_from_EPIC-KITCHENS, 2020], but we failed to fix their bugs. Consequently, further exploration of this egocentric kitchen Dataset will be left to future research.

6.2.4 Secondary Image Processing

In the background section of this chapter, we revised many traditional machine vision techniques. Nevertheless, one common problem with these methods is that they are susceptible to image sizes and scene contexts. For example, applying thresholding-based colour detection over an entire image may result in various outcomes, as a scene may contain multiple regions that share the same colour. To address the issue, we introduce secondary image processing, or bounding-box-based image processing.

Essentially, the idea is to use object detection results to generate a smaller image that is then processed to retrieve more specific pixel information related to a particular task. We present four techniques we used and discuss how they could help Amiga to do kitchen tasks. The primary library used in this section is the OpenCV library [Itseez, 2015]. Conversion between ROS messages and OpenCV formats can be achieved using the [ROS_CV_Bridge, 2010 package].

Colour Detection: Colour recognition is one of the basics of image processing. The idea is straightforward; we specify a range of RGB intensities (lower and upper threshold) we wish to extract from an image. The process can be accomplished with OpenCV. First we convert images to HSV spaces HSV using `cv2.cvtColor` function. Then we apply `cv2.inRange` to extract a colour mask and apply it back to the original image using `cv2.erode` and `cv2.dilate` functions. The use of colour detection is task-orientated. Suppose, for example, the KitchenNet detects a blender, one of the classes it trains on, and we would like Amiga to launch the blender by pushing a button. If the button happens to have a distinctive colour, we can use this method to retrieve its 2D pixels from a cropped blender image and precise the 3D coordinates.

Edge Detection: The Canny edge detector evaluates the intensity gradient of an image to detect edges. The gradient intensities are computed using pixel-wise derivatives and applied with *NMS* or *hysteresis thresholding* procedures in order to suppress spurious responses. We implemented this operator using OpenCV's `cv2.Canny` function after grey-scaling and `cv2.Gaussian-Blur`. Figure 6.4 shows an instance where the edge of a cropped box was detected via this method. In this case, the noises introduced in the bounding boxes were eliminated, leaving clear edge pixels. Mapping the edge pixels to 3D coordinates assists Amiga in carrying out activities such as packing into boxes or mopping a table (which are both detectable by KitchenNet). To

raise above, bounding boxes derived from object detection are not perfect, but edge detection is capable of refining them.

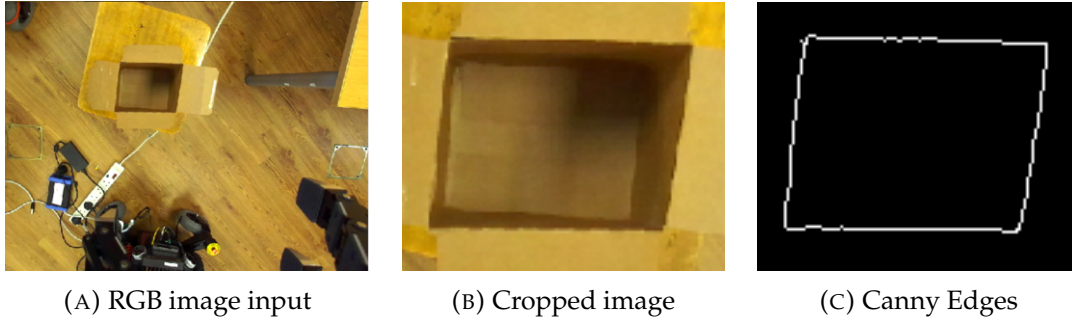


FIGURE 6.4: Edge detection on Amiga's view in real world

Circle Detection: The *Hough transform* is a transformation from image space to parameter space. Given an input set of edge points, it produces a parametric model. Every edge point is a vote for a potential model for a specific parameter space. Given a circle parameterisation,

$$(x - a)^2 + (y - b)^2 = r^2$$

we can fit circles into an image. This is implemented using OpenCV's `cv2.HoughCircles` function. Figure 6.5 shows an example where we applied this technique and detected the circle perimeter of a bowl. These pixels can then be converted to 3D coordinates and assist Amiga's tasks, such as stirring and mixing the content in this bowl. Naturally, this concept can be extended to fitting other shapes, such as rectangles, depending on the use case.

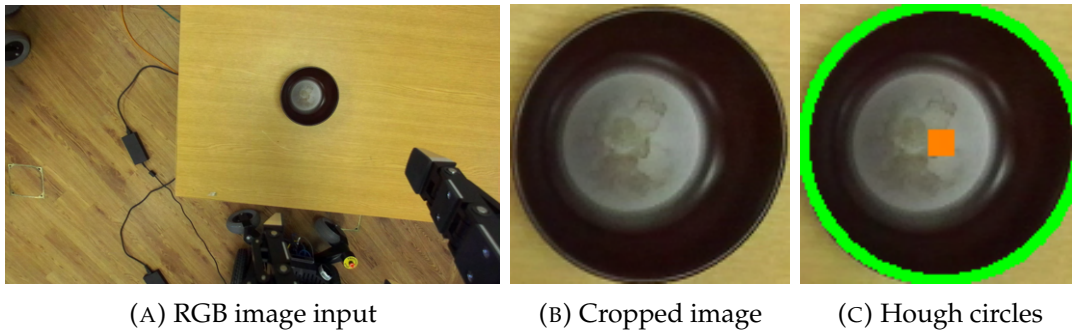


FIGURE 6.5: Circle detection on Amiga's view in real world

Interest Point Detection: As already discussed in this chapter's background, representative visual features can be extracted using feature descriptors. We can follow these methodologies and extract interest points from a cropped image. Figure 6.6 shows an example where a SURF descriptor was applied to a beer in Gazebo using the `[find_object_2d, 2016]` package. This approach generates a set of key point pixels for highly textured regions of the object.

We can later convert them to 3D interested coordinates, averaging and returning a single coordinate representing the most textured point on the object. The point can then be used in our 3 DoF grasping module to represent a precise 3D location for grabbing, which we will discuss later.

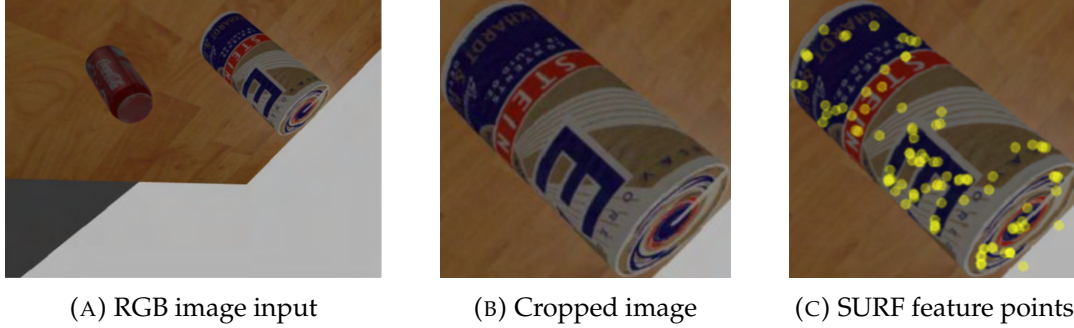


FIGURE 6.6: Interest point detection in Amiga simulation

The literature contains numerous methods for image processing and presented here is only a tip of the iceberg. However, the concept is inspirational. For an image $f(x, y)$, we detect and crop an object's image $f'(x, y)$:

$$f'(x, y) = f(x, y) | B_{obj}^6 \quad \forall B_{obj} \subseteq \{x \times y\}$$

where B_{obj} is a bounding box. Next, we use secondary image processing methods⁷ to obtain one or more interested pixels, $\{u, v\}$, such that:

$$\{u, v\} \subseteq B_{obj} \subseteq \{x \times y\}$$

The pixel set $\{u, v\}$ is then converted to a set of region-of-interest 3D coordinates (in the camera frame) as according to Section 6.2.2 and published for the manipulation, grasping and navigation modules.

6.2.5 Object Point Cloud

The concept developed in Section 6.2.2 implies the possibility of conversions between depth images and point clouds given camera information. We used the [depth_image_proc, 2020] package for this purpose. Figure 6.7 demonstrates the synthesis of a cropped RGB image and a cropped depth image into an XYZRGB point cloud. The difference between colours is due to rendering issues during the visualisation of saved point cloud files. The use of the object point cloud is summarised as below:

1. Pass to the dimension estimation node, which we will discuss in the next section
2. Pass to the 6-DoF grasping module, which will be discussed later

⁶ $f(x, y)$ restricted to a bounding box of an object

⁷Note in many image processing methods the image also needs to be converted to grayscale first

3. What follows has not been implemented yet. Still, the idea is to pass to the pose estimation node, where an iterative closest point algorithm is used to compare it with a mesh model from the database, hence deciding the object's pose. In addition, since this object point cloud is limited to partial views, we could complete its shape with multiple views.

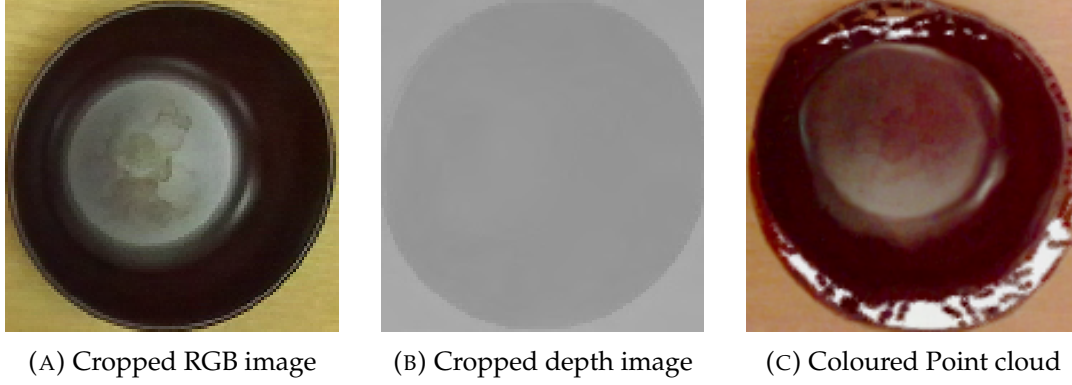


FIGURE 6.7: Conversion of cropped RGB and depth images to XYZRGB point clouds

As visible in Figure 6.7c, an object's point cloud may contain ground as a result of cropping out of the bounding boxes, which is undesirable. We propose methods to remove the ground in the next section.

6.2.6 Plane Detection

In this section, we implement methods to remove the ground from a point cloud (or equivalently a depth map), which we refer to as plane detection.

Suppose we have a point cloud, $PC(X, Y, Z)$, the task now is to detect and segment out the portions representing the ground, $PC_{ground} \subseteq PC$. For each point $x_g, y_g, z_g \in PC_{ground}$, the following must hold:

$$ax + by + cz + d = 0$$

Therefore, an intuitive solution is to fit a plane equation to all points $(x, y, z) \in PC$. This can be achieved using Hough Transform as discussed in Section 6.2.4 or Least squares and find a, b, c , which minimize the sum of squared perpendicular distances:

$$\mathcal{L}_{a,b,c} = \sum_{i=1}^n (ax + by + cz + d)^2$$

However, the problem is that due to the unavoidable noises from the sensors, the set PC may contain many outliers with distance $\gg 0$, and hence minimising via Least squares is not a robust solution. Instead, we use the RANSAC algorithm introduced in Preliminary 2.5.

Initially, the implementation was attempted with the Python wrapper for PCL [pythonPCL, 2018], but it was no longer maintained after 2018. In order to use it, we had to install earlier versions of PCL, but the sources were highly

unorganised, and we were unable to use it. As an alternative, our first successful implementation was based on the Open3D [Zhou, Park, and Koltun, 2018] library in python.

Figure 6.8 displays the result applied to the real-world point cloud data taken from Amiga’s arm camera view, using RANSAC parameters of distance_threshold as 0.02, sampling number as 3, and the number of iterations as 1000. The inliers of a plane obtained by the algorithm are visualised in red as in Figure 6.8b.

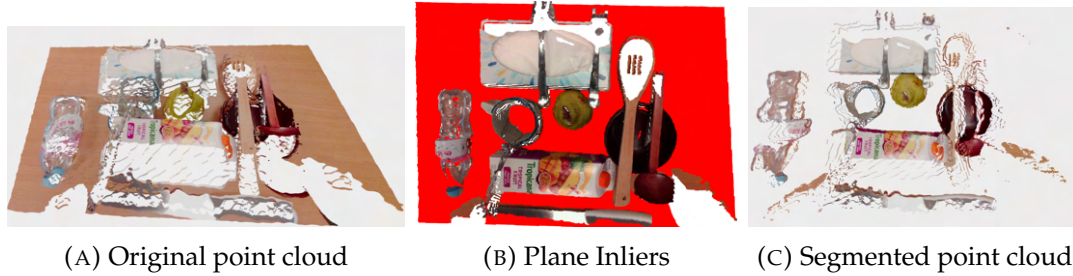


FIGURE 6.8: Plane detection and segmentation

In spite of the algorithm’s success from the perspective of accuracy, its computation time was far from satisfactory, taking 5 seconds, which was clearly unsuitable for topic publishing purposes. Naturally, we thought of improving the speed by lowering the number of iterations. After many experiments, it was discovered that the number of iterations could be reduced at most to 100 without significantly affecting the model’s performance. Still, the computation would require around 2 seconds, which was not ideal.

In order to improve our plane detection node’s performance, we re-written the algorithm in C++ using the PCL library [Rusu and Cousins, 2011]. While this was an effortful undertaking, it was rewarding as well – the plane detection could now publish segmented point cloud messages at a frequency of at least 15 HZ, depending on the input size.

Furthermore, apart from segmenting out the ground contained in an object/region point cloud, plane detection is also helpful for obstacle avoidance, where planes are often treated as non-collision objects. In this context, a scene might include multiple planes, such as the ground and a table. In order to address this issue, we came up with a modified version of the algorithm, which detects multiple planes by continuously applying RANSAC and segmentation until the ratio between the remaining points and the original data is smaller or equal to a new threshold parameter min_percentage. At termination, all plane inliers are copied to a new cloud for publishing. We will discuss this in more detail in Section 7.2.9.

In a nutshell, the plane detection developed in this section is used as follows:

1. Absorbed in the cropped-depth-images-to-pointcloud node ⁸ and publishing the ground-free object point cloud in real-time

⁸The absorption is because it does not make sense to use separately a node subscribing to an object/region point cloud with ground, segmenting and publishing it again

2. Modified to multiple plane fitter, subscribing to the raw point cloud stream from the camera and publishing the point cloud of all the planes inside for obstacle avoidance purposes

6.2.7 Region Detection

So far, most of what we have established has been presumed on the success of the object detection model. Nevertheless, this may not always be the case – sometimes, the detector may not produce any results at all. Therefore, we propose region detection, or foreground detection, in which a single region bounding box is determined based on RGB/depth images, thus decreasing the reliance of Amiga perception on object detection.

Consider an image $f(x, y)$, now our task is to determine $f'(x, y)$ such that:

$$f'(x, y) = f(x, y)|_{B_{\text{foreground}}}^9 \quad \forall B_{\text{foreground}} \subseteq \{x \times y\}$$

As this topic is a vivid field of research in computer vision, our discussion is primarily within this project's scope. Consider a situation where many undetectable objects are put on the ground or a table, and Amiga has a top-down view. We attempt to propose a region detector that detects a cluster region. The idea developed here lays the foundation for Amiga's object-agnostic deep grasping and is used to perform tasks such as clearing a table off all objects.

We attempted two approaches based on two different sensor sources – RGB images and depth images.

RGB image method: An intuitive idea to separate foreground and background is to apply a cut-off threshold to the image histogram. However, this only works if the image pixels have a uniform distribution, while in reality, they are more likely to be in Gaussian distribution. So we can fit two Gaussian models to the foreground and background pixel sets – this is the basic idea behind the Gaussian Mixture Model (GMM) [Reynolds, 2009]. While we experimented with this method (the results are shown in Appendix A.2.4), it was not employed in the end primarily because of two reasons:

- Computation time: since our RGB images are of quite high resolutions, applying iterative clustering algorithms directly on three channels of pixels would take more than 3 seconds
- Noises: the direct outputs from the algorithm generally have high variances and will need further processing, especially when a scene contains both a ground and a table

⁹ $f(x, y)$ restricted to a bounding box of the foreground

Depth image method: The problem can be solved faster by using depth images, which only contain one channel. Initially, the approach was based on the results presented in the previous section. According to our constraint settings, the background consists primarily of planes. A similar plane fitting method can be applied to depth images, as discussed in Section 6.2.6. It was, however, decided that re-implementing our node to work with depth images would not be worth the time¹⁰. In its place, a much more straightforward solution was proposed. Picture a depth image of objects on a table – naturally, the pixels belonging to the ground/table would share similar depth values compared to those of objects. Hence we could compute the derivatives and extract the gradient-intensive regions. An edge map can be generated using OpenCV's `cv2.GaussianBlur` along with `cv2.Sobel`, followed by thresholding the histogram with `cv2.calcHist`, and proposing a bounding box based on the leftover active pixels. However, as Amiga navigates, the scene may sometimes contain both a table and a ground (which the GMM method handles poorly). We hence proposed a new algorithm to address this issue:

1. Compute derivatives from raw depth images $g(x, y)$. Locate all pixels whose gradient intensity is larger than a threshold (table-ground) and below another (gripper-ground). Fit and compute four line edges, which model the boundaries between the table and the ground
2. Use boundary edges to generate two new depth images, $g_1(x, y)$ (values inside set to 0) and $g_2(x, y)$ (values outside set to 0)
3. Use the procedures described previously to propose two regional boxes
4. Map the boxes back and select the one with high activity (fewer zeros)

Figure 6.9 shows the result of our implementation. We have successfully detected a region-of-interest box in a scene where both ground and table exist.

Here, the errors are caused by the gripper appearing in the image view after switching the camera from L515 to ZED 2. Re-printing a camera mounting clamp would be much work, so we will leave it for a later date.

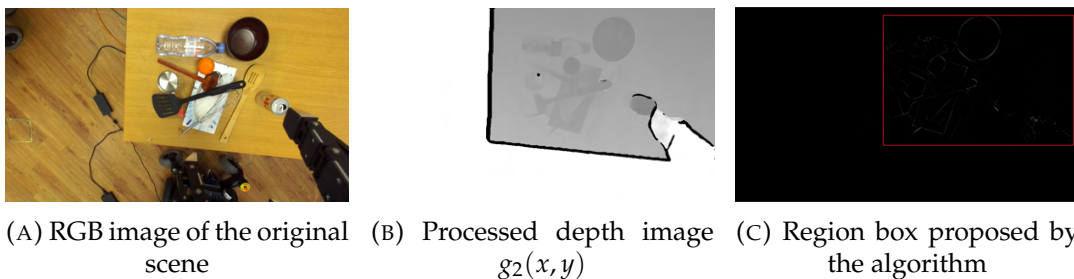


FIGURE 6.9: Region detection based on computing gradients

¹⁰C++ developing process can be rather tedious

Overall, this algorithm turned out to be quite successful under our setting¹¹. Sometimes there were random errors, but they only contributed to a small boundary box and hence have little impact on the final results.

6.2.8 Dimension Estimation

In this section, we describe the construction of the dimension estimation node on top of ground-free objects or region point clouds. The idea developed here lays the foundation for Amiga manipulation’s obstacle avoidance sections in which the dimensions of objects are estimated, fitted into primitive shapes (cube, cylinder, etc) and then added as collision objects.

In the beginning, we used a brute-force cut-off method to iterate the whole point cloud and decide $x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}$ which allows modelling of a cuboid. However, this method suffers seriously from noise, often overestimates and returns unreasonable results. Although it is possible to use statistical methods [statistical_outlier_removal, 2020] to remove outliers, the computation time would become prohibitive as the size of the point cloud increases.

Instead, we propose an alternative method based on geometry. Figure 6.10 illustrates the geometrical concept behind it.

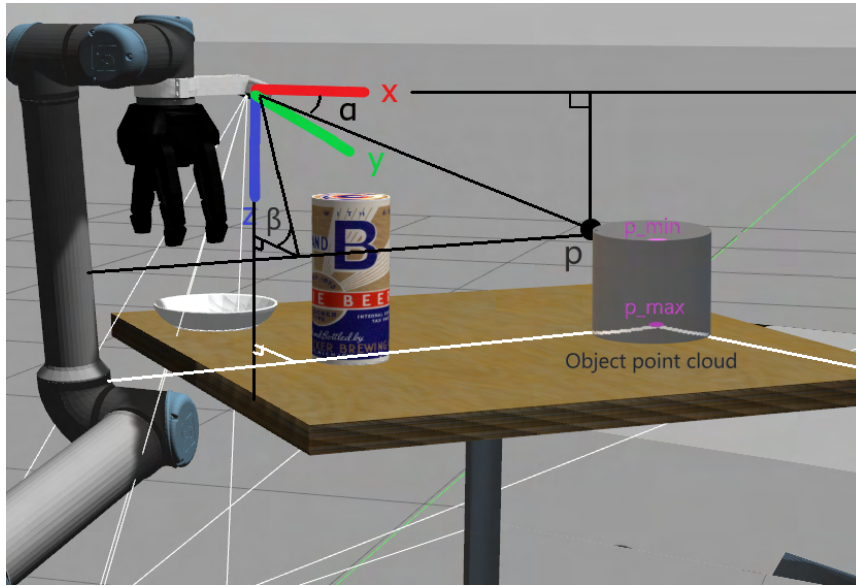


FIGURE 6.10: Geometric concepts behind dimension estimation

Suppose we have a camera with the coordinate frame shown in the figure where red, green, and blue represent the x , y , and z -axis, respectively. Suppose we have an arbitrary point p on a cylinder object point cloud with 3D coordinate (x_p, y_p, z_p) . Under this setting, z is simply the perpendicular distance to the camera, and the actual distance d from p to the camera can be computed using the Pythagorean theorem:

¹¹A Top-down viewed scene where clustered objects spawn on either (or both) on the ground or (and) the table

$$d = \sqrt{x_p^2 + y_p^2}$$

Moreover, we define the angle α as the angle p made horizontally along the x -axis. This gives:

$$\alpha = \text{atan2}(z_p, x_p)$$

Likewise, we define another angle β as the angle p made vertically along the y -axis. This gives:

$$\beta = \text{atan2}(z_p, y_p)$$

It is now possible to use β to estimate the dimension of the cylinder object point cloud. In Figure 6.10, a cylinder object point cloud $PC(x, y, z)$ with $x, y, z > 0$ is being captured by the camera. There exists two points $p_{min}, p_{max} \in PC$ such that the angle β is minimised and maximised, respectively. The locations of p_{min} and p_{max} have been visualised in Figure 6.10, where p_{max} lies on top of the cylinder and closer to our eyes and p_{min} lies on the bottom of the cylinder, invisible to our eyes.

A more general illustration is shown in Figure¹² 6.11 below, which depicts p_{min}, p_{max} in $PC(x, y, z)$ for $y < 0$ and $y > 0$.

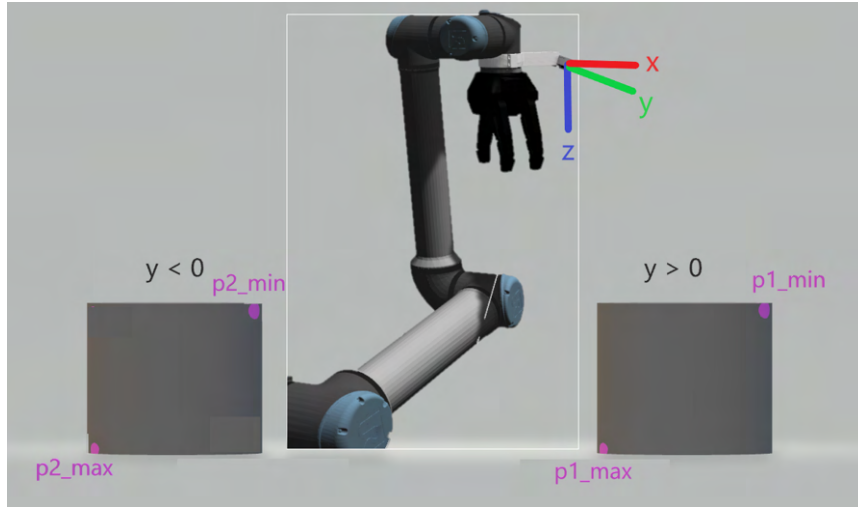


FIGURE 6.11: Illustration of geometric concepts behind the dimension estimation node in more general case

Consider $z > 0$ as $z < 0$ would be absurd, we have therefore four cases:

1. The point cloud is in front and to the right of Amiga $\leftrightarrow x > 0$ and $y > 0$
2. The point cloud is in front and to the left of Amiga $\leftrightarrow x > 0$ and $y < 0$
3. The point cloud is behind and to the right of Amiga $\leftrightarrow x < 0$ and $y > 0$
4. The point cloud is behind and to the left of Amiga $\leftrightarrow x < 0$ and $y < 0$

¹²Note again the two objects are assumed to be cylinders in the Figure

Figure 6.12 presents the graph of $\text{atan2}(z, y)$ over z/y . We are now to minimise/maximise $\beta = \text{atan2}(z_p, y_p)$ for the above four cases.

- Case 1 lies in the first quadrant of the graph. To minimise β , we should minimise z_p/y_p , which entails minimising z_p and maximising y_p . This corresponds to the point $p1_min$ in Figure 6.11. Similarly, $p1_max$ is the point which maximises β
- Case 2 lies in the second quadrant of the graph. To minimise β , we should minimise z_p/y_p , which aligns with case 1. The maximiser $p2_max$ and minimiser $p2_min$ are drawn in Figure 6.11.
- Case 3 coincides with case 1 since β is independent of x
- Case 4 coincides with case 2 since β is independent of x

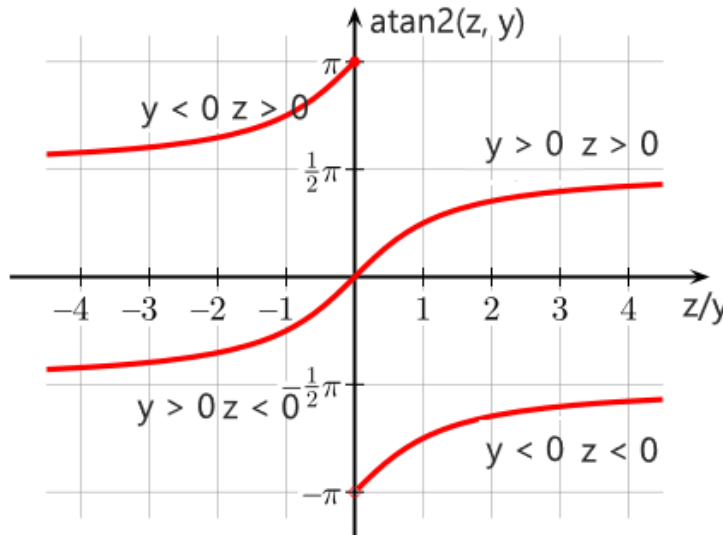


FIGURE 6.12: Graph of $\text{atan2}(z, y)$ over z/y

Similar analogies can be applied to maximise and minimise the angle α . We can iterate the whole point cloud and find in a total of four points. Compared to the original brute force cut-off implementation, this method is faster and less prone to noises-led overestimation because instead of relying on a single x, y, z value to vote for six candidate dimension estimation points, it utilised the ratio z/y and z/x to propose four.

Granted, the algorithm established so far presumes on cylinder objects where a unique pair of p_{min} and p_{max} exist. A cuboid shape, for example, may contain many such points. However, the idea extends. For a cuboid, the only difference among the minimisers and maximisers for β is the x -coordinate; hence we can average them to estimate dy, dz . Similarly, we use the minimisers and maximisers for α to estimate dx, dz . The two dz s can then be averaged again to produce a final dz . Moreover, as the shape of the object

becomes complex, there could be, for example, two β minimisers with different y, z (but $\beta = \text{atan2}(z_p, y_p)$ equals and is minimised anyway). To account for these cases, all the minimisers and maximisers for β and α are stored in a vector and averaged. The use of these averages also alleviate the impact of noise.

6.3 Architecture Summary and Comments

In this section, we summarise the work we have done in regards to Amiga's perception in a kitchen context. With the previous sections in this chapter setting the scene, we have essentially established a framework for Amiga Perception, as shown in Figure 6.13.

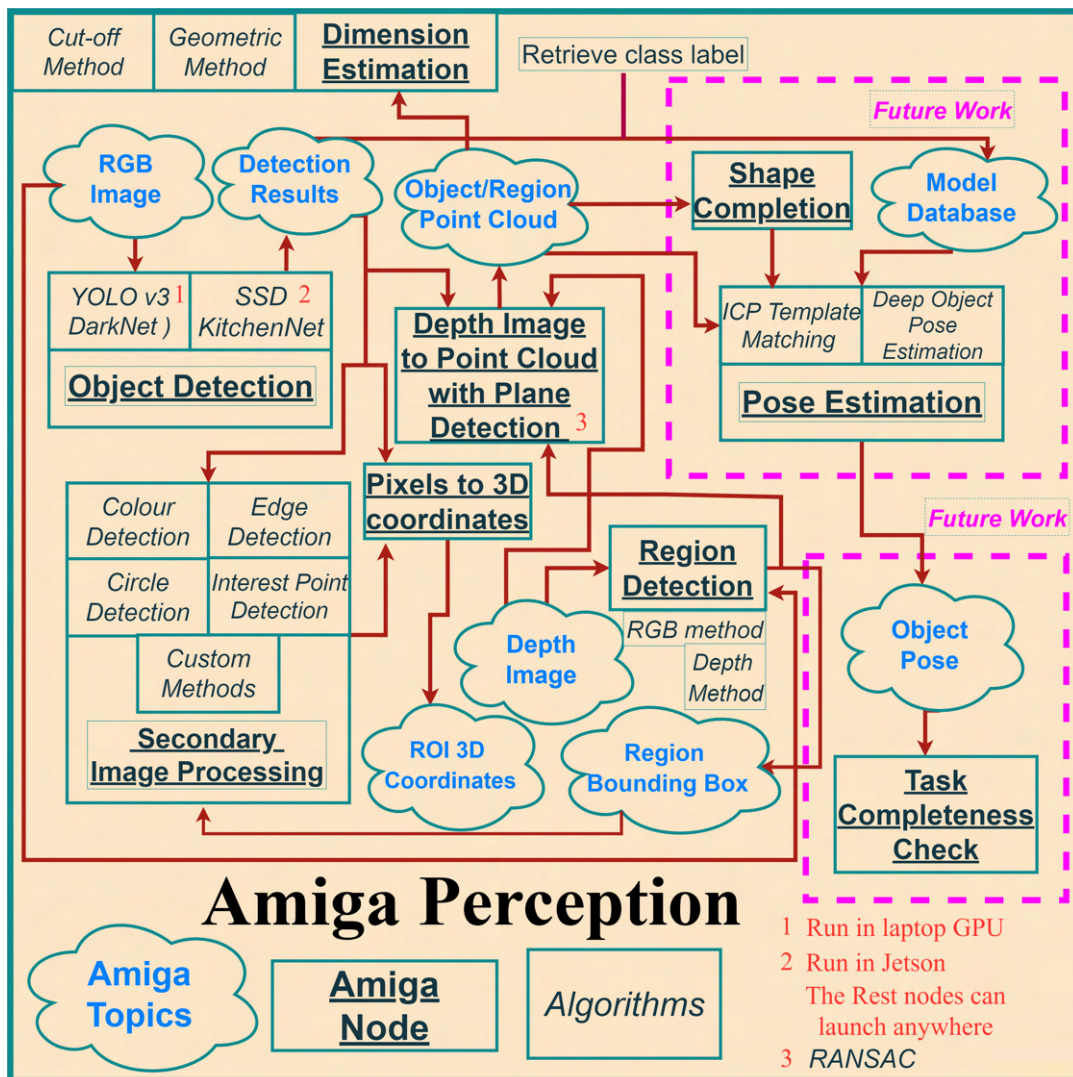


FIGURE 6.13: Amiga Perception in its entirety. Blue clouds represent data topics, underlined rectangles denote software nodes, italicized rectangles indicate algorithms used, and pink-dashed boxes specify future work

Perception is a crucial aspect without which Amiga cannot perform any useful kitchen tasks with autonomy. Let us look at the nodes in the figure above and quickly review our work:

1. **Object detection:** replicated three object detection nodes, concluded their underperformance in detecting kitchen objects and proposed our KitchenNet, which includes 37 categories trained on 100K images. The detection results are published in real-time at 200 HZ.
2. **Secondary image processing:** included a series of image processing algorithms to retrieve more task-specific pixels of interest in an object/region. In our case, circle detection is used for stirring; edge detection is used for packing and interest point detection for 3-DoF grasping.
3. **Region detection:** proposed as an alternative to object detection to detect a foreground region of interest. Initially implemented with GMM based on RGB images and later devised an improved gradient boundary algorithm based on depth images.
4. **Depth image to Point cloud:** Utility node to publish object/region point cloud based on object/region detection results and depth maps.
5. **Pixels to 3D coordinates:** developed to convert interested pixels detected into 3D interested coordinates using aligned depth maps. Although reading is also possible via point cloud directly, it is experimentally tested to be much slower and unstable.
6. **Plane detection:** used to remove the planes contained in an object/region point cloud. Implemented a modified RANSAC node in C++ to detect/extract all the planes with at least 15 HZ.
7. **Dimension estimation:** estimate the dimensions of the ground-free object/region point cloud so that they can be added to manipulation's collision scenes. Proposed a brute-force cut-off-following-outlier-removal method and devised an efficient geometric method which reduces the effect of noise without having to remove outliers (computation-intensive).
8. **Shape completion (Future work):** Object/region point cloud from a single viewpoint is only partial, complete it by fusing multiple views.
9. **Pose estimation (Future work):** for all objects that can be robustly detected, build a 3D mesh model and register into a database. Retrieve the detection results and employ template matching to determine the object's pose. Alternately use deep learning methods.
10. **Task Completeness Check (Future work):** using the estimated object pose information, devise algorithms that recognise whether an object manipulation task has achieved the intended results.

The framework is naturally interchangeable and extensive. For example, one might argue that the ROI interested coordinates can also be retrieved directly from an object/region point cloud. We can build a node and connect the two by devising appropriate data processing techniques. However, It is prudent not to process an extensive collection of point clouds directly because it requires substantial computation.

Moreover, we should also point out that what we have proposed is essentially an engineering framework that must take into account both the theories and the applicability. There are plenty of limitations and opportunities for improvement, which we will address later.

In the next few chapters, we will delve deep into how the builds we created in this chapter connect to Amiga's manipulation, grasping and navigation.

7 Amiga Manipulation

7.1 Background and Module Design

Motion planning is a computational problem of solving a sequence of feasible configurations that move a target from the source to a desired destination. In this chapter, we refer to motion planning as explicitly the movement of a robot arm. A successful vision-guided motion planning consists of four components: *inverse kinematics*, *hand-eye calibration*, *collision checking* and *path planning*. Let us discuss each of them in turn.

7.1.1 Configuration, FK and IK

First, we define the *joint configuration* q as a set of generalised joint coordinates:

$$q = \{q_j, j \in [1 \dots N]\} \quad (7.1)$$

where q_j is the joint coordinate and N is the number of joints. Each of the coordinate can be either an angle (for *revolute joints*) or a length (for *prismatic joints*). Next, we define the *configuration space*, $\mathcal{C}$, which represents the space of all possible configurations:

$$q \in \mathcal{C} \subset \mathbb{R}^N$$

Further, we define the *work space*, $\mathcal{W}$, as the space for all possible end-effector poses ξ_E :

$$\xi_E \in \mathcal{W}$$

For a 6-DoF robot arm such as UR10e, the following holds:

$$\mathcal{C} \subset \mathbb{R}^6, \quad \xi_E \sim (x, y, z, \phi, \theta, \psi), \quad \mathcal{W} \subset \mathbb{R}^3 \times \text{SO}(3) \quad (7.2)$$

Forward kinematics (FK) computes the work space coordinates of the robot given the joint configuration as input. In our case:

$$q \sim (q_1, q_2, \dots, q_6) \mapsto \xi_E \sim (x, y, z, \phi, \theta, \psi) \quad (7.3)$$

The kinematics equation contains a chain product of successive coordinate transformation matrices from the base link to the end-effector link.

By contrast, *Inverse kinematics* (IK) is the reverse operation which computes joint configurations such that desired work space coordinate is reached:

$$\xi'_E \sim (x', y', z', \phi', \theta', \psi') \mapsto q' \sim (q'_1, q'_2, \dots, q'_6) \quad (7.4)$$

As opposed to forward kinematics, inverse kinematics generally cannot be characterized by closed-form expressions as the mapping from the configuration space to the workspace is nonlinear. The As a result, an IK problem may have no solution, multiple solutions, or even an infinite number of solutions. Figure 7.1 shows an example in a 2-DoF arm case where two solutions (q_1^A, q_2^A) and (q_1^B, q_2^B) exist. As the number of joints and degree of freedoms increase, IK problems can become very challenging to solve.

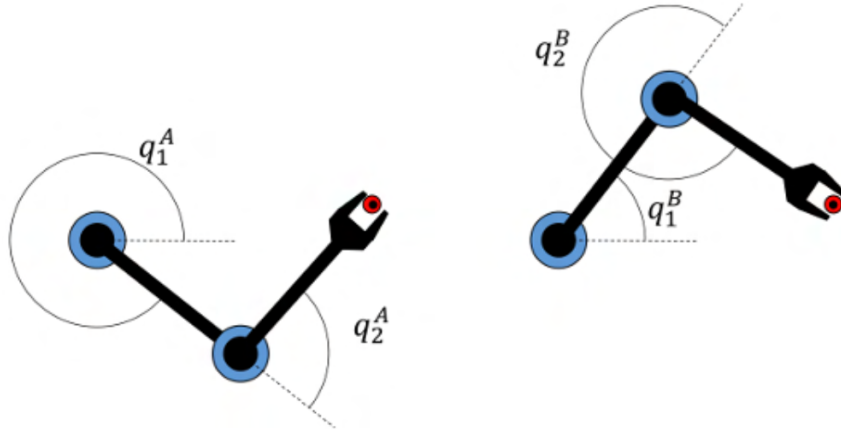


FIGURE 7.1: Two IK solutions (q_1^A, q_2^A) and (q_1^B, q_2^B) in a 2-DoF robot arm [Hauser, 2020]

A variety of techniques have been proposed to solve the IK problems over the years. They can be divided into two categories, analytical methods and numerical methods.

Let us rewrite the FK mapping in Equation 7.3 as:

$$f(q) = \xi_E \quad (7.5)$$

In the analytical approach, the Equation 7.5 is mathematically inverted into a closed-form expression via symbolic manipulations. On the other hand, the numerical methods start with initial guesses and optimise the errors instead:

$$\min \|f(q) - \xi_E\| \quad (7.6)$$

Remark

While the analytic methods compute solutions rapidly once the equations are derived, the derivation process can be very difficult and tedious. Furthermore, robots with different kinematic structures must be derived separately. On the other hand, numerical methods are more general approaches which account for arbitrary robot structure and work spaces free of any structure-specific mathematics apart from forward kinematics and Jacobian computations. Consequently, their use is preferred in software packages that integrate multiple types of robots. Although many researchers have addressed the UR10's kinematics, such as in [Hawkins, 2013], few projects are available for reference, especially in the UR10e case. By contrast, many mature libraries for numerical methods are attainable. In this respect, the numerical method is

less risky and easier to implement. To begin with, we will use the standard jacobian-based numerical solver from the [Kinematics_and_Dynamics_Library, 2021] library. Compared to other solvers such as the Trac-IK solver [TRAC Labs_Robotics, 2021], whose solution is not as stable despite high success rates, and the IKFAST solver [Diankov, 2010], which despite being fast, suffers from problems where a choice must be made among multiple possible solutions, the KDL solver produces more robust solutions and is more straightforward to use. Nevertheless, one commonly acknowledged problem is that it is more likely to fail and has relatively low efficiency.

After evaluations and depending on the performance, we will then consider switching to more advanced solvers.

7.1.2 Hand-Eye Calibration

The URDF file introduced in Preliminary 2.1 already specifies the transformations amongst various links and frames in the robot. As well as the standard links, one must take into account the custom-assembled links, such as the camera sensor links. In robotics, the problem of deciding the transformation between an end-effector and a camera formulates the *hand-eye calibration problem*.

It is common for problems of this nature to involve two situations: eye-in-hand and eye-to-hand. While the former depicts the scenario where the camera is mounted on the wrist of the robot end-effector, the latter suggests the camera is fixed somewhere else in the world. For a mobile robot like Amiga, an eye in the hand setup is more common since, for the other case, a recalibration must be conducted every time the mobile base moves.

The hand-eye calibration problem was initially formulated by Shaheen Ahmad in [Shiu and Ahmad, 1989], following which a number of techniques were developed till today. Figure 7.2 demonstrates an eye in hand calibration scenario, where a camera is mounted on a gripper. Using extrinsic calibration techniques developed in Preliminary 2.3, we obtain the camera-world transformation A_i . Moreover, from the joint angle values, the kinematic chain leads to the robot's base to gripper transformation B_i . For one pose, we observe two transformations unknown: robot base to the object and camera to gripper X . To eliminate the base to object transformation, we need two poses, which a motion can obtain. Suppose the arm moves from a first pose to a second pose, the following relationship holds:

$$\{b\} \mathbf{T}_{\{g_1\}\{g_1\}} \mathbf{T}_{\{c_1\}\{c_1\}} \mathbf{T}_{\{o\}} = \{b\} \mathbf{T}_{\{g_2\}\{g_2\}} \mathbf{T}_{\{c_2\}\{c_2\}} \mathbf{T}_{\{o\}} \quad (7.7)$$

where $\{a\} \mathbf{T}_{\{b\}}$ denotes the homogeneous transform from frame a to frame b , $\{g_1\}$, $\{g_2\}$ and $\{c_1\}$, $\{c_2\}$ denote the first and the second poses of the gripper and camera, $\{b\}$ and $\{w\}$ denote the base and the object frame, which remain unchanged following an arm motion. Equation 7.7 can be rearranged to produce:

$$\{b\} \mathbf{T}_{\{g_2\}\{b\}}^{-1} \mathbf{T}_{\{g_1\}\{g_1\}} \mathbf{T}_{\{c_1\}} = \{g_2\} \mathbf{T}_{\{c_2\}\{c_2\}} \mathbf{T}_{\{o\}\{c_1\}} \mathbf{T}_{\{o\}}^{-1} \quad (7.8)$$

Further ,

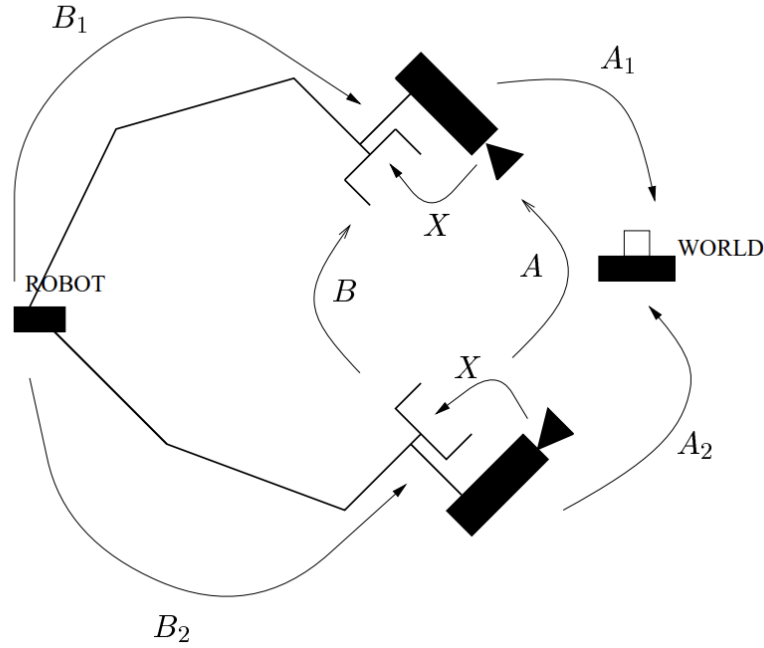


FIGURE 7.2: Illustration of an eye in hand calibration setup where a camera is mounted on a gripper

$$\begin{aligned}
 A &= \{b\} \mathbf{T}_{\{g_2\}}^{-1} \{b\} \mathbf{T}_{\{g_1\}} \\
 B &= \{c_2\} \mathbf{T}_{\{o\}\{c_1\}} \mathbf{T}_{\{o\}}^{-1} \\
 X &= \{g_1\} \mathbf{T}_{\{c_1\}} = \{g_2\} \mathbf{T}_{\{c_2\}}
 \end{aligned}$$

We have formulated the well-known hand-eye calibration Equation:

$$AX = XB \quad (7.9)$$

In Equation 7.9, while A is very accurate to millimetre scope, B is not as accurate. Therefore instead of solving $AX - XB = 0$, one can formulate and solve for an unconstrained optimisation problem instead:

$$\min \|AX - XB\| \quad (7.10)$$

the solution can be estimated quite accurately by performing N^1 motions such that $A = (A_1, A_2, \dots, A_N)$ and $B = (B_1, B_2, \dots, B_N)$

Remark

In this project, an eye-in-hand calibration is necessary. More specifically, we need to estimate the transform between the end effector link, `amiga_arm_tool0` and the optical frame of the camera. The transformation errors will have an adverse effect on the motion planning module's performance.

¹Empirically the errors plateau after about $N = 15$ samples are collected

7.1.3 Collision-aware Path Planning

In Section 7.1.1 we defined the joint configuration q as a set of generalised joint coordinates. Now we define an abstract *configuration space*, or *C-space*, such that on the C-space $\mathcal{C}$, each point p represents a particular configuration q :

$$q = (q_1, q_2, \dots, q_N) \mapsto p \in \mathcal{C} \subset \mathbb{R}^N \quad (7.11)$$

Now suppose the presence of obstacles, which partitions the configuration space into two sets, $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$ ², which represent the set of free space and obstacle space, respectively.

With the above, the *path planning* problem can be formulated as a problem of moving from q_{start} to q_{end} fully in $\mathcal{C}_{free}$ space.

While it is possible to transform obstacles into their C-Space equivalents and obtain $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$, the two sets become difficult to compute as the dimension of the spaces grows. By contrast, detecting collisions is relatively easier. To test if $q \in \mathcal{C}_{free}$, one can use FK to determine the robot's geometric position, followed by *collision checking* to determine possible collisions with the environment.

Deciding if a robot at a particular configuration q is in collision against a given environment usually requires rather complex operations, including CAD models of both the environment and robot. Many software packages exist for this purpose, such as the FCL library [Pan, Chitta, and Manocha, 2012]. Some of the regular tasks include:

1. Collision detection: determining if the two models overlap and, if so, identifying which triangles overlap.
2. Distance computation: determining the minimum distance between two models or the distance between the two closest points.
3. Tolerance verification: checking if two models are closer or farther than a tolerance distance.

Let us get back to the motion path planning problems. The general rule is that there is no universal motion planner that can be applied to all motion planning problems [Lynch and Park, 2017]. We compare and summarise some of the popular methods below:

- **Complete methods:** these planners establish on an exact representation of $\mathcal{C}_{free}$. Although mathematically and algorithmically complex, these techniques are not useful for many high-degree-of-freedom applications (such as ours). Examples such as [Liu, Trinkle, and Milgram, 2004] and [Milenkovic, Sacks, and Trac, 2013] only involve 2 DoFs.
- **Grid methods:** these planners discretise $\mathcal{C}_{free}$ into a grid and search it for a path from q_{start} to a point on a grid in the goal region. Despite the fact that these methods are easy to employ and can yield optimal

²Note joint limits are also treated as obstacles in $\mathcal{C}$

results, the amount of memory required to store the grid and the time it takes to search it increases exponentially with the number of dimensions of $\mathcal{C}_{free}$. A real-time evaluation of these methods [Saranya et al., 2014] suggests their inapplicability to high-dimensional problems.

- **Nonlinear optimisation:** these planners essentially convert the motion planning problems to nonlinear optimisation problems. The path is represented by certain design parameters³ The problem hence becomes a minimisation problem subject to obstacle and goal constraints. A distinctive example is the Covariant Hamiltonian optimisation for motion planning (CHOMP) [Ratliff et al., 2009]. While most high-dimensional motion planners divide trajectory generation into distinct planning and optimisation stages, this algorithm capitalises on a covariant gradient, and functional gradient approaches the optimisation stage to design a motion planning algorithm based entirely on trajectory optimisation [CHOMP_Planner, 2021].
- **Sampling methods:** these methods sample a s from $\mathcal{C}$ and evaluate if $s \in \mathcal{C}_{free}$. Moreover, they also devise a function to devide the “closest” previous sample $s_{previous} \in \mathcal{C}_{free}$ and employ a local planner to connect, or move the new sample s_{new} from the previous sample, $s_{previous}$. This procedure establishes a graph denoting the feasible trajectories of the robot. An example is the PRM algorithm, which builds a connectable road map $R \in \mathbb{R}^1$ with K nodes such that $\mathcal{C} \in \mathbb{R}^N \mapsto R \in \mathbb{R}^1$. The graph is then searched for a path, typically with A^* algorithm. An example of a PRM roadmap is shown in Figure 7.3 below.

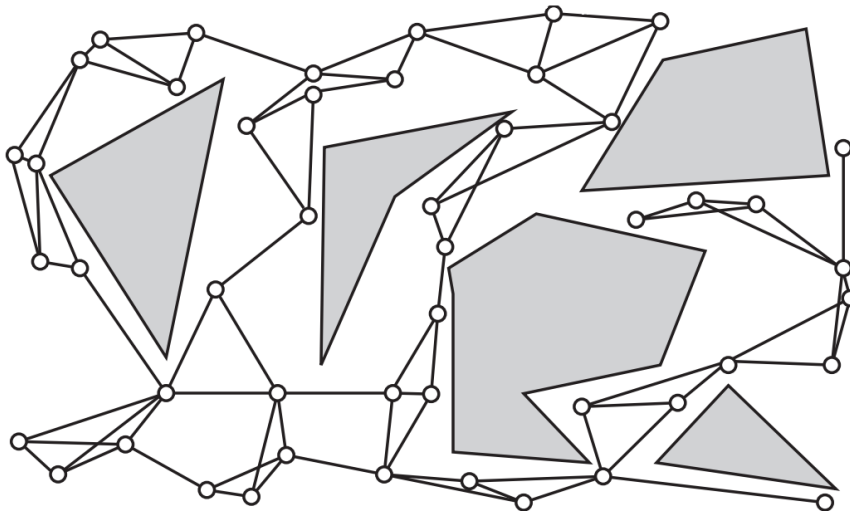


FIGURE 7.3: An example PRM roadmap with $\mathcal{C} = \mathbb{R}^2, K = 3$ [Lynch and Park, 2017]

³For example, the coefficients of a polynomial or a Fourier series.

Although sampling methods' solutions may most likely be suboptimal, they are easy to implement and capable of addressing high DoF motion planning problems, hence becoming a popular trend.

Remark

Since in this project we are dealing with high DoF planning problems, only optimisation and sampling methods mentioned above are feasible. When it comes to implementation, despite the CHOMP algorithm, another available optimisation-based planning package is STOMP (Stochastic Trajectory Optimisation for Motion Planning) [Kalakrishnan et al., 2011], which features stochastic gradient-free optimisation and requires no gradient information during planning. On the other side, most SOTA sampling-based methods are collected in the Open Motion Planning Library [Şucan, Moll, and Kavraki, 2012]. OMPL provides a meta-optimisation algorithm entitled AnytimePathShortening, which repeatedly runs multiple planners in parallel interleaved with path shortcutting and path hybridisation, two techniques employed to optimise a solution path locally. In spite of the fact that this strategy has not yet been proven as an optimal approach, it is often effective in achieving near-optimal results in practice and hence will be adopted in our project.

7.1.4 MoveIt Planning Framework

Having covered the basics of motion planning, it is clear that several independent components are available in the form of software packages. Integrating them into robot-driven systems is not a trivial process. For example, the planners in OMPL are essentially abstract and have no concept of a robot. This means one must manually configure OMPL and provides the back-end for it to work with problems in Robotics. The procedures can become even more tedious if later we decide to integrate another library. The MoveIt framework [Coleman et al., 2014] is designed specifically for this.

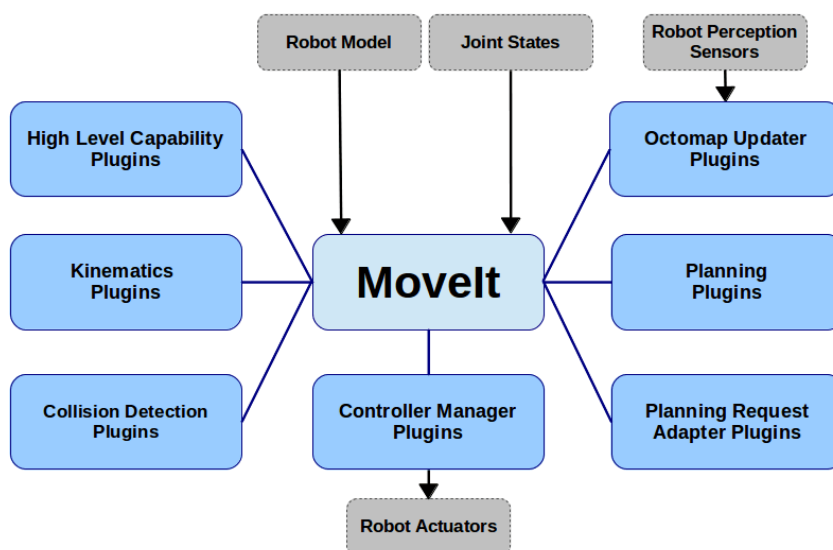


FIGURE 7.4: A conceptual view of MoveIt's interface

MoveIt is an integrated developing platform comprised of a series of motion planning, operating control, and 3D perception plugins, as shown in Figure 7.4. Many software packages we went through exist in the form of plugins. Below are some of the plugins we are most concerned about and the options MoveIt already supports. Our selection has been marked with ticks, and the reasons for this have been outlined previously. Other options (not supported by MoveIt but can be customised) remain open before evaluation.

- **Kinematics plugins:** KDL (✓), Trac-IK, IKFAST
- **Planning plugins:** OMPL (✓), CHOMP, SBPL⁴
- **collision detection plugins:** FCL (✓), PCD, Bullet⁵
- **Octomap plugins:** OctoMap library (✓ / will be discussed later)

The dynamically feasible trajectories generated by MoveIt are then sent to robot controllers, which then drive the motors either in real-world or in simulation.

7.2 Implementation

7.2.1 Arm Configuration

The first thing we need to do is to configure the UR10e arm to the ROS network, which can be achieved using the [UR_ROS_Driver, 2021] package. The driver consists of a C++ library that abstracts UR10e's interfaces and a ROS driver on top of that.

We configured the driver by writing a launch script starting a node which we named `ur_hardware_interface`, and launched the `ur_robot_driver`. The parameters needed to pass include the arm's ip⁶, communication ports, input/output recipe file and a .yaml file specifying calibrated kinematics for each joint. This calibration result was generated using the driver package's `calibration_correction.launch`, without which the end-effector positions might be off by a few centimetres. To set up ROS communication, we wrote and loaded to ROS parameter servers the `amiga_ros_controllers.yaml`, configuring the ROS controllers we introduced in Chapter 5. Other nodes, such as the controller spawners, stoppers and joint state publishers, were written similarly to the ones used in Amiga simulation.

To control the arm via ROS, use the action server on:

```
/scaled_pos_joint_traj_controller/follow_joint_trajectory
```

⁴A generic set of motion planners using search-based planning

⁵Bullet features continuous collision capabilities, which guarantees that no collision can occur during the transition between two discrete robot states with the environment. However, unlike FCL, Bullet does not calculate all individual contact points for a shape but only the point of deepest penetration.

⁶The IP assigned was "10.0.3.7" (recall Amiga's network in Figure 3.9)

The e-series of the UR arms are associated with a tablet, as introduced in Chapter 3.1, which builds several sockets to control the arm. This is not elegant because every time we launched Amiga, we had to use the tablet. As a result, we decided to merge the tablet features to our `ur_hardware_interface` node by accepting several external scripts broadcasting the dashboard services. Some of the frequently used examples are:

- `dashboard/power_on` : power on the robot motors
- `dashboard/unlock_protective_stop` : dismiss the protective stop
- `dashboard/restart_safety` : used when the arm gets a safety fault as a result of collisions or our intervention (pushing the emergency buttons)

More features, such as the teach-and-repeat paradigms, still require the tablet and remain open for future integration.

7.2.2 Manipulation Interface

Now that the arm can move and be controlled manually by us, it is time to implement motion planning. As stated previously, we will be using MoveIt to implement the manipulation module. To begin with, it is necessary to set up MoveIt for Amiga. We used MoveIt - Noetic as it corresponds to our versions.

MoveIt provides a convenient setup assistant to generate default configuration files for the plugins automatically. Different supported plugins such as kinematics solvers can be switched by downloading relevant packages and modifying the configuration file. However, the unsupported ones may require more intervention, such as the analytic IK solvers provided by the UR industrial repository [UR_Industrial, 2021].

Moreover, in addition to a URDF file, MoveIt requires another SRDF (Semantic Robot Description Format) file, which complements the URDF and specifies joint groups, default robot configurations, and additional collision checking information transforms that are required to specify the robot's pose completely. It is enough to configure the arm groups for our purposes, as the gripper will be controlled independently.

While it is true that we only wish to use MoveIt to control the Amiga's arm, Amiga is essentially not just an arm, as evident from its URDF files, which inevitably include a frame 'odom' due to the presence of the wheelchair base. The situation becomes even worse given that we will need two versions of MoveIt, one from Amiga's laptop, which use Dockers and one from an external developer's laptop, which prefers not to use Dockers because it is constantly under active production.

Getting Amiga to smoothly run MoveIt under our desired setting⁷ is a non-trivial task.

⁷Can handle both only-arm and full robot situations and address the inconsistency between dockerised MoveIt running in Amiga laptop and debug/dev versions running in the developer's laptop

Only-arm vs full-robot: To begin with, let us address the only-arm and full-robot dilemma. In essence, we need to be able to use MoveIt in both cases similarly. We addressed this situation by considering different cases depending on whether Amiga manipulation and navigation are used:

1. Always publish the robot state of all joints in URDF
2. No manipulation/navigation: publish fixed joint states for all the joints
3. No manipulation but navigation: directly launch the navigation stacks
4. Manipulation without navigation: only publish the fixed joint values for the base and publish a fixed static transform from odom to dummy⁸
5. Manipulation with navigation: launching navigation stacks afterwards (roslaunch has execution orders) should overwrite existing settings

Dockerised vs debug version: This step took us more than a week. For some reason, we could not use the services, particularly the compute cartesian services provided by the Move group node launched on Amiga's laptop, and there were no errors in the terminal. The returned trajectories are simply empty. A straightforward solution would be using dockers⁹ as well in the dev laptop; however, this will mess around the production environment where numerous different packages are installed. In addition, the main drawback of dockers is that it is just not suitable for a productive environment. It would not make sense to build a new image (which can take arbitrarily long sometimes) every time we wish to modify the source code. Another solution would be just developing on Amiga's laptop, but we would prefer not to take the lab's laptop home, transferring numerous files and messing up its environment in production. MoveIt is itself under active development and regular release, but its version controls are generally poor. There are plenty of ways to install MoveIt, such as apt-get, wstool and build from source, but they do not interface with each other despite being in the same version. After many continuous attempts, including re-installments of ROS and transferring different packages from the container to our laptop, we came up with a stable, consistent solution by modifying both the docker files and hacking the local environment.

From the above two processes and up to now, we eventually achieved a manipulation interface¹⁰ via MoveIt between Amiga and any external laptops that have MoveIt installed. However, the interface was essentially still MoveIt, and we would need to develop more features based on the this platform. Once all features are implemented, we will export them into a new docker image and run it on Amiga's laptop directly.

⁸The joint we defined in URDF where the arm_base connects the wheelchair

⁹The docker image used in Amiga's laptop contains both ROS and MoveIt

¹⁰The communication in Amiga network is achieved by assigning an IP to the user laptop as in Figure 3.9)

7.2.3 Manipulation Server

Now that the Interface is ready, we can use MoveIt to control Amiga. In MoveIt, script-based control is attained via a `moveit_commander` node, which abstracts a robot commander class, a `MoveGroupCommander` class and a `PlanningSceneInterface` class. Each time we want to use MoveIt to program Amiga to perform a robot action, we need to prepare a script and use `roscpp` to initialise class objects and specify Amiga-specific parameters.

Spamming multiple instances of `moveit_commander` nodes in Amiga network is not elegant; moreover, many kitchen-related robotics actions can be divided into sub-tasks and resolved independently. Based on these observations, we propose Amiga manipulation server, a MoveIt-based hierarchical service architecture used to systematically analyse and solve manipulation tasks in the context of a robotic kitchen assistant.

During the next few sections, we will follow a button-up approach to build this architecture. With increasing sophistication, the level of interaction between this architecture and Amiga perception (Chapter 6) cultivates.

7.2.4 Building Blocks

Recall from the background section of this chapter that motion planning is a computational problem of solving a sequence of feasible configurations that move a target from the source to the desired destination. The interfaces provided by MoveIt in [MoveIt_Interface, 2022] already cover many aspects. However, we will need to investigate and configure the parameters such that they can be applied to Amiga. For instance, the `computeCartesianPath` interface returns a path in the `odom` frame for Amiga, a non-static volatile frame in which simple straight-line trajectories become complex curves. We fixed these issues by explicit modifications on the generated trajectories and wrapped the common-used functions into abstract manipulation blocks for our manipulation server.

Another basic function is the transformation amongst different coordinate frames. While this can be attained by `tf.lookupTransform`, it was experimentally tested to be less stable. Our implementation using `tf2_ros` is shown in Appendix A.3.1.

The manipulation server is essentially a ROS package; after defining the required `srv` messages, modifying `CMakeLists.txt` and compiling in `catkin` it can be launched, passing some self-explanatory parameters in a launch file similar to the following:

```
<node name="amiga_manipulation_server" pkg="manipulation_server"
  type="manipulation_server.py" output="screen">
  <param name="simulation" value="false" />
  <param name="goal_position_tolerance" value="0.001" />
  <param name="goal_orientation_tolerance" value="0.001" />
  <param name="pose_reference_frame" value="base_link" />
  <... ...>
</node>
```

The initial services we provide are as follows:

Service Function	Input	Output
Get current pose	Empty	Pose
Get joint values	Empty	Radian values
Plan to target	Pose with frame	Bool success
Plan joints	6 radian values	Bool success
Plan cartesian	pose waypoints ¹¹	Bool success

TABLE 7.1: Available services in the manipulation server so far

More specifically:

- Get current pose: retrieve the current pose $(x_c, y_c, z_c, rx_c, ry_c, rz_c, rw_c)$ of the end-effector `Amiga_arm_tool0`
- Get joint values: retrieve joint configuration $(q_1, q_2, \dots, q_6)$ from the arm
- Plan joints: plan and go to a joint configuration $(q'_1, q'_2, \dots, q'_N)$
- Plan to target: given a pose $(x_t, y_t, z_t, rx_t, ry_t, rz_t, rw_t)$ in a reference frame $\mathcal{T}$, plan the end-effector there w.r.t `base_link`
- Plan cartesian: specify a list of waypoints (poses) for the end-effector to go through in Cartesian space w.r.t `base_link`

These basic motion planning blocks can be called and in any devices connected to Amiga network via `rosservice` call or `rospy.ServiceProxy`. For example, to lift the arm in Cartesian space by 20 cm, simply do:

```
x, y, z, rx, ry, rz, rw = getCurPoseProxy.call()
waypoints = [Pose(x, y, z + 0.2, rx, ry, rz, rw)]
planCartesianProxy.call(waypoints)
```

Moreover, it will be helpful to define certain robot poses. For our purposes, we defined a navigation-safe pose (arm contracted within the wheelchair base to avoid collisions), a grasping-home pose (gripper and camera facing to the ground to have top-down views) and a grasp-front pose (gripper and camera facing forwards to have front views). A set of figures describing these poses are put in Appendix A.3.2 since many have appeared before. These poses are communicated to MoveIt via defining and naming them in the SRDF file specifying a group state. We included them in the manipulation server such that driving to an example grasp pose is as simple as:

```
rosservice call /amiga_manipulation/go_to_home_grasp
```

¹¹With optional resolution as input, if none default to 0.01

7.2.5 Virtual-effect Translator

Now that we can drive the end-effector to the desired pose, we should probably consider the arm-gripper offset. Furthermore, as the project progressed, we were inspired to generalise the concept by proposing the notion of the virtual-effect translator.

Considering grasping as an example, the "end-effector" we wish to drive probably lies in the centre of the gripper rather than `amiga_arm_tool0`. To resolve this issue, one must examine the transformation between them. Consider Figure 7.5. Due to the nature of the assembly, the gripper itself cannot pivot around either the x-axis (showing in red) or the y-axis (showing in green), and its rotation around the z-axis (showing in blue) can only be achieved via rotating `amiga_arm_tool0`. Having said that, the transformation only concerns a translation, which can be retrieved from the manual for the gripper and, in this case, 20 cm. Moreover, suppose the origin point in the `virtual_effector` frame and a sphere centred around it (indicated with the white circle); the points on the sphere's surface essentially span a set of feasible locations for `amiga_arm_tool0`. The solution becomes unique once the orientation of the sphere centre is fixed.

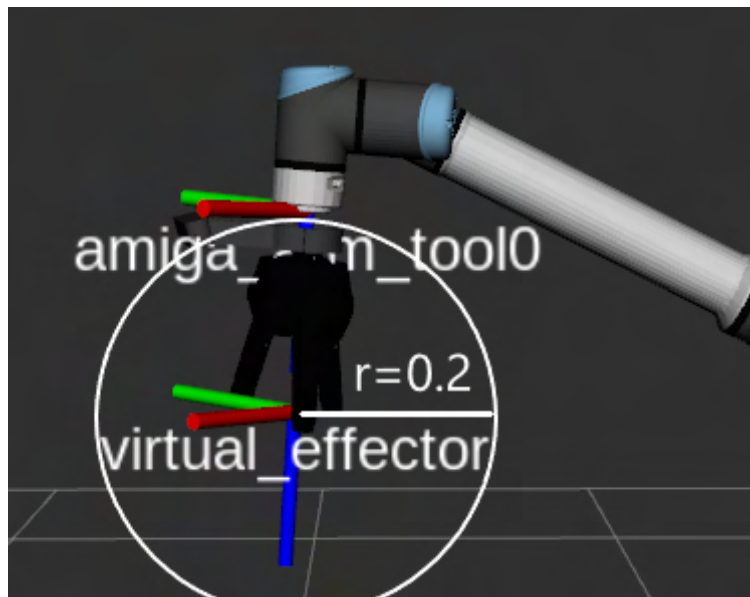


FIGURE 7.5: The concept of the virtual effector. Only translation is involved.

Until now, one without a solid knowledge of kinematics might attempt to define this virtual effector link in the URDF and modify the server to regard it as the end-effector during planning. This approach is problematic because, in addition to tarnishing the URDF, such a link does not exist and should not be checked for collisions. In addition, it is generally possible to have multiple virtual effectors. If, for example, Amiga holds a pen and we wish it to draw on paper, the tip of the pen can serve as a virtual effector. Spamming all virtual effectors in the URDF would be inappropriate.

Instead, we propose two solutions:

IK method: (✗) recall from the background that the FKs are essentially kinematic transformation chains to drive the end-effector to a reference coordinate. Having established the transformation between the end effector and virtual effector, one may regard the virtual effector as the end result of a kinematic chain and use IKs to solve for a set of joint configurations such that the virtual effector lands at a particular pose¹². In MoveIt, something similar occurs all the time – the only difference is that the IK is calculated manually on this occasion. MoveIt’s compute IK services could be employed for this purpose, or we could use external solvers instead.

Analytical method: (✓) granted, using IKs followed by planJoints seems a bit complicated. As far as the gripper is concerned, all virtual effectors consist of only translations and hence it will not be difficult to resolve the transformation via geometry. Suppose a desired virtual effector pose $pose_{vir} = (x, y, z, \phi, \theta, \psi)$ in a reference frame and a unique corresponding end_effector pose $pose_{ee} = (x', y', z', \phi, \theta, \psi)$ in the same reference frame. The rotations are fixed and what we want is essentially the point $p_{ee} = (x', y', z')$, which can be computed directly as in Equation 7.12:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \mathbf{R}_{33}(\phi, \theta, \psi) \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \quad (7.12)$$

where t_x, t_y, t_z represents the relative translation between the virtual effector and the end-effector and $R_{33}(\phi, \theta, \psi)$ represents the 3D rotation group we introduced in Preliminary 2.2. Once the positions are computed, inferring $pose_{ee} = (x', y', z', \phi, \theta, \psi)$ is trivial. Essentially we have established a mapping from a desired virtual-effector’s pose to its corresponding end-effector’s pose w.r.t a reference link:

$$pose_{ee} \mapsto pose_{vir}$$

Our concept naturally extends, suppose now we have a trajectory of waypoints for the end-effector Amiga_tool0, $s_{ee} = (pose_{ee}^1, pose_{ee}^2, \dots, pose_{ee}^n)$, we could map all of the poses in the set and achieve:

$$s_{ee} \mapsto s_{vir}$$

Combined with Amiga perception’s dimension estimation (Section 6.2.8) and the subsequent Amiga grasp, one could perform many exciting tasks such as grasping an object, estimating the transformation to an identified virtual effector and making it move along a predefined trajectory. An unsuitable metaphor is analogous to an extensive invisible hand for Amiga.

¹²Our planJoint service would then be used to reach the result configurations

7.2.6 Grasp Executor

Now let us verify the effectiveness of our virtual-effect translator by building a grasp executor. Consider a cloth-grasping task in Figure 7.6. The green region represents the obstacle and the lower visualised frame, `grip_base` link is a 6-DoF grasp pose generated by Amiga Grasp’s deep grasp algorithms, all of which will be presented later. For now, suppose we have been given a desired 6-DoF grasp pose in the `base_link` frame for the gripper to reach.

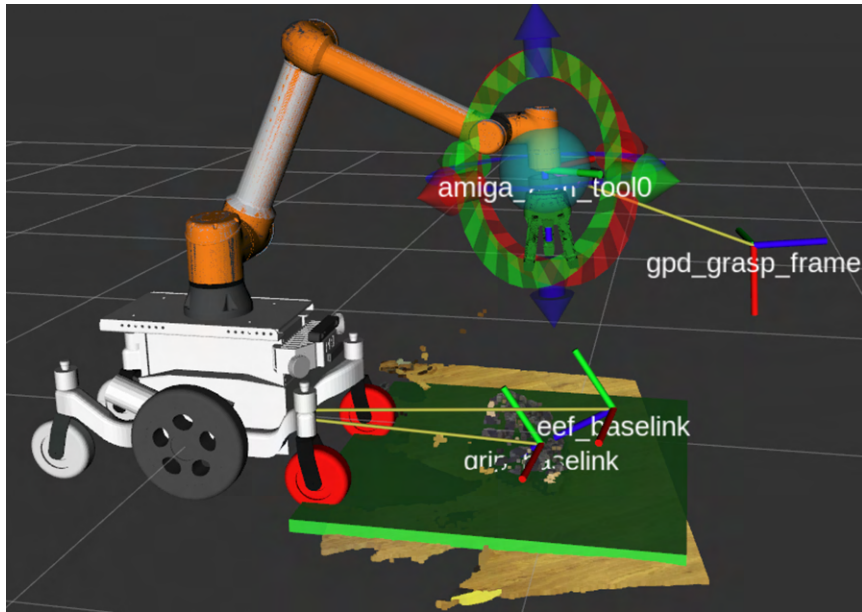


FIGURE 7.6: Experiment verification of virtual-effect translator

Pseudocode like:

```
grasp_pose = amigaDeepGraspGenerator.call()
# [0, 0, 0.2] = [tx, ty, tz] w.r.t eef frame (gripper offset)
eef_pose = virtualEffectTransProxy.call(grasp_pose, [0, 0, 0.2])
```

can give us the `eef_base` link in the Figure (from the `grip_base` link). Moreover, in some cases, we may not want to plan to the grasp pose directly as it may invoke a collision. This can be true in Figure 7.6 since the cloth is not rendered as the obstacle. Instead, we may want to plan to a pre-grasp pose and then plan to the final pose. In 6-DoF cases, the pre-grasp offset adjustments can be rather complicated due to the presence of rotation. Nonetheless, this task can be solved easily by our translator like:

```
# [0, 0, 0.15] = [tx, ty, tz] # tz is essentially the distance
pre_pose = virtualEffectTransProxy.call(eef_pose, [0, 0, 0.15])
# recall section 8.2.4
planToTargetProxy.call(pre_pose, 'base_link')
planToTargetProxy.call(eef_pose, 'base_link')
```

The gripper should move to the pre-grasp (15cm away from the grasp pose) and then move to the target grasp pose in consecution. For an actual grasp, we only need to close the gripper and move the gripper somewhere else.

7.2.7 Viewpoint Adjuster

In this section, we build another block to prepare for the participation of Amiga perception – the viewpoint adjuster.

Recall that previously we have defined a set of default poses for Amiga, including a front-view and top-down pose. Now picture Amiga is in the second pose (similar to Figure 7.6), facing down and trying to detect objects. From a single viewpoint, some of the objects may not be detected.

One may think of some line trajectories to move the camera closer, but in essence, they are just translations and will not affect the viewpoints. Another solution would be to define the different viewpoint poses manually in the SRDF, naming them and using the manipulation server to drive them there on demand. However, this will require a lot of manual definitions, and the method itself is neither general nor elegant. Instead, we propose the viewpoint adjuster. This idea is inspired by Figure 7.5. The main principle is to generate several poses around a reference frame (in our case, the camera, but it can also be any particular objects) by sampling spherical coordinates.

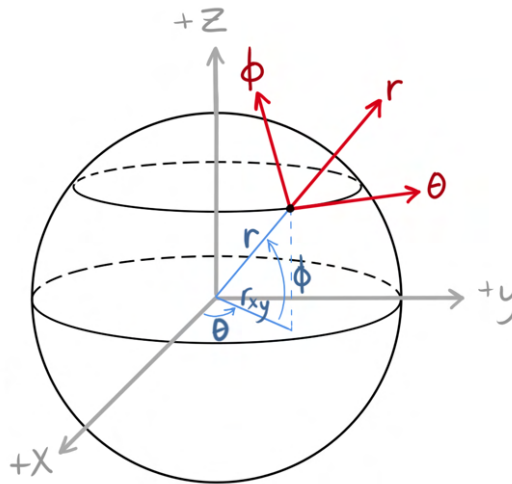


FIGURE 7.7: Spherical coordinate systems

Consider Figure 7.8, which depicts a spherical coordinate system. The following holds:

$$x = r \sin\theta \cos\phi$$

$$y = r \sin\theta \sin\phi$$

$$z = r \cos\theta$$

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = \cos^{-1}(z/r)$$

$$\phi = \tan^{-1}(y/x)$$

Now our goal is to sample some poses on the sphere's surface, such as the one with axes in red colour in Figure 7.8. The approach we devised was based on a uniform sampling of the sphere by diving the sphere layers L , the longitude ϕ and the latitude λ into equidistant segments. Therefore, given a user-defined L, λ, ϕ , we can sample in total of $L \cdot \lambda \cdot \phi$ poses in a sphere of radius r via Algorithm 1:

Algorithm 1 Spherical Coordinate Sampling

Require: L, r, λ, ϕ

Ensure: $L, r, \lambda, \phi > 0$

```

1: function SAMPLEPOSE( $L, r, \lambda, \phi$ )  $Poses \leftarrow []$ 
2:   for  $i \leftarrow 1$  to  $L$  do
3:     for  $j \leftarrow 1$  to  $\phi$  do
4:       for  $k \leftarrow 1$  to  $\lambda$  do
5:          $\theta \leftarrow \pi \cdot \left(\frac{90}{\lambda+1}\right) \cdot \frac{(\phi-1)-j}{180}$ 
6:          $\phi \leftarrow \pi \cdot \left(\frac{360}{2 \cdot \phi}\right) \cdot \frac{i}{180}$ 
7:          $x \leftarrow L \cdot r \cdot \sin\theta \cdot \cos\phi$ 
8:          $y \leftarrow L \cdot r \cdot \sin\theta \cdot \sin\phi$ 
9:          $z \leftarrow L \cdot r \cdot \cos\theta$ 
10:         $R \leftarrow \theta \cdot \sin(-\phi)$ 
11:         $P \leftarrow \theta \cdot \cos(\phi)$ 
12:         $Y \leftarrow 0$ 
13:         $Poses \leftarrow [x, y, z, R, P, Y]$ 
14:       end for
15:     end for
16:   end for
17:   return  $Poses$ 
18: end function

```

Figure 7.8 demonstrates the results of our algorithm applied around the centre of the camera frame. The point_centre frame is decided initially by computing the normal vector of the sphere. In practice, we might need to introduce several offset parameters for each $[x, y, z, R, P, Y]$ generated by the algorithm.

A set of poses essentially specify a trajectory. Hence, we could either solve an optimised trajectory analytically over these generated pose sets or just use the manipulation server to plan to each destination.

Now recall our initial objective – to adjust viewpoints if KitchenNet cannot detect objects well. Some pseudocode would be:

```

While KitchenNet.detections == None:
    #[2, 0.4, 4, 3] = L , r, \lambda, \varphi

```

```
Poses = sampleSpheticalPoseProxy.call([2, 0.4, 4, 3])
for Pose in Poses:
    # recall section 8. 24; essentially transform these
    poses to base_link and execute if plan found
    planToTargetProxy.call(pose, 'zed2_left_camera_link')
    time.sleep(5)
```

We could use this method to evaluate the performance of kitchen object detectors because it unravels more information rather than simply feeding different objects to the arm camera. Of course, one would argue that this approach has an inherited limitation: the objects may be detected during the path from a sampled pose to another. However, the arm cannot stop in the middle; hence, by the time it lands on a new viewpoint, the object may no longer be detected. Improvements can be performed in the future¹³.

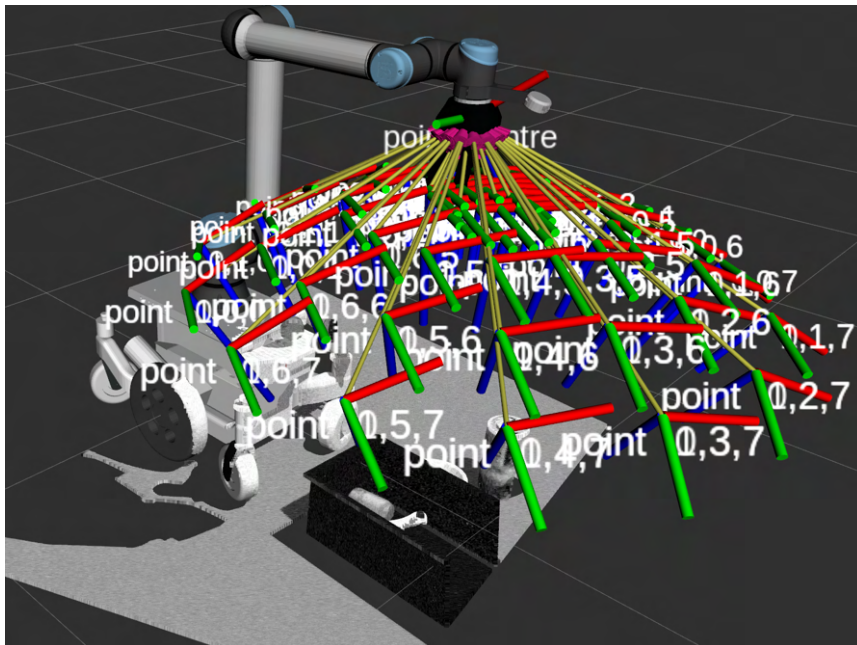


FIGURE 7.8: A set of sampled spherical poses centred around the objects in the camera frame

7.2.8 Vision Participation

Now that we have some basic building blocks, one may start collaborating with perception. To begin with, let us address the hand-eye calibration problem. We used the [MoveIt_Calibration, 2020] package for this purpose. The ideas are introduced in Section 7.1.2 and the mechanism is similar to the simulation cases in Appendix A.1.2, so we will not present them any further. The only difference was that we needed actually to print one calibration grid. We

¹³Maybe optimise a trajectory or simply use simple visual servoing rather than planners (under obstacle-free assumption)

calibrated twice for L515 and ZED 2; the results can be specified manually in the URDF, or broadcasted using a static transform publisher:

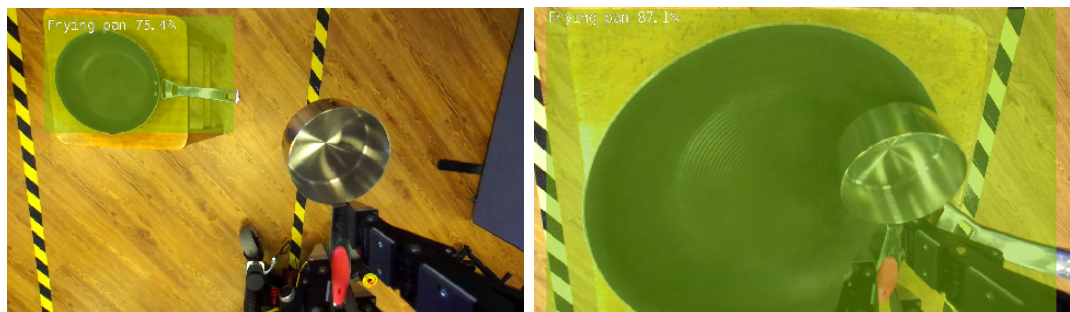
```
<node pkg="tf2_ros" type="static_transform_publisher"
  name="camera_link_broadcaster"
  args="-0.157587 -0.00233397 -0.126455
  0.991751 -0.00156368 0.127601 0.0120261
  amiga_arm_tool0 zed2_base_link" />
```

Further, we have to know the stored frames of topics published by Amiga perception. Echoing the topic message directly is messy, instead, we run:

```
rostopic echo -n1 --noarr /zed2_node/point_cloud/cloud_registered
```

Compared to L515, where both depth images and point clouds are published in the optical frame, the output of the above command suggests for ZED 2, while the depth images are stored in the optical frame, the point cloud data is stored in the `zed2_left_camera_frame`. This used to cause confusion.

An example of simple vision participation is shown in Figure 7.9, which demonstrates a vision-guided placing task. In this case, Amiga perception identifies and locates a frying pan via our kitchenNet, publishing a ROI 3D coordinate (recall Section 6.3 from the last chapter), and we employ the manipulation server to drive it to that position.



(A) KitchenNet detecting a frying pan

(B) Manipulation server driving the gripper

FIGURE 7.9: An example of Vision-guided placing task

Pseudocode like

```
_x, _y, _z = rospy.wait_for_message('/amiga_perception
  /frying_pan/roi_3d_point', Vector3)
x, y, z, rx, ry, rz, rw = getCurPoseProxy.call()
waypoints = [Pose(_x, _y, _z, rx, ry, rz, rw)]
_waypoints = virtualEffectTransProxy.call(waypoints, [0, 0, 0.2])
planCartesianProxy.call(_waypoints)
```

is sufficient for our purpose. The idea extends; we can now interact with Amiga perception to perform any sort of placing tasks. Moreover, the tea-spoon is one of the objects that Amiga grasping (we will see in the next chapter) with any orientation on a plane (4,5-DoF grasping) can actually grasp, and hence we can use the system to grasp it, drive it on the frying pan, and pour the content out (we will discuss how to pour later).

7.2.9 Obstacle Avoidance

In our discussions thus far, a great deal has been ignored in regards to obstacles. Now that perception is present, Amiga should be able to recognise obstacles and avoid them as it moves.

As far as the collision detection is concerned, MoveIt's planning scene can access multiple different collision detectors. Two abstract classes, `collisionWorld` and `collisionRobot`, are responsible for self-collision checks and checks for robot against the environment, respectively. Since `collisionRobot` is tracked via URDF, we mainly discuss `collisionWorld`. We propose and implement four kinds of collision detection techniques.

Primitive obstacles The default support from MoveIt is to add primitives to scenes, which can include boxes, spheres, cylinder and so on. We integrated these features to our manipulation server such that something like:

```
# centres of the obstacle; no orientation in this case
pose.position = [0.82, 0.16, -0.15]
collisionSceneAddProxy.call("ground", pose, [1.0, 1.0, 0.035])
```

will upload to the `collisionWorld` scene an obstacle named "ground", which centres around the pose and has dimension $1 \times 1 \times 0.035$ ¹⁴. The result of the above corresponds to the green plates in Figure 7.6. Amiga perception's dimension estimation node (Section 6.2.8) returns exactly the information required for this feature.

Octomap – all are obstacles: As introduced in the background 7.1.4, MoveIt supports the Octomap library [Hornung et al., 2013], which implements a 3D occupancy grid mapping approach, supplying data structures and mapping algorithms in C++. Thanks to MoveIt, to enable this feature, we only have to install the package and configure a plugin named `occupancy_map_monitor/PointCloudOctomapUpdater`. Essentially, this works by using the Octomap library to implement a node subscribing to a particular point cloud and publishing its converted grids. MoveIt then adds the grids to the planning scenes. Figure 7.10 demonstrates Amiga equipped with such collision knowledge in the planning scene. In Figure 7.10a, the source is from the arm camera, and in Figure 7.10b, the source is from multiple cameras from both the arm and

¹⁴Length, width and height in metres

the base. We registered the point clouds as seen by multiple cameras using the [autoware_ai_perception, 2020] package. Granted, no matter how many cameras we use, there exist blind spots. We hence argue that if needed, we could instead use the RPLIDAR A1M8 Laser Scanner to give 360° Octomaps. With enough resolution, collision-free movement of Amiga's arm can be assured (although one must address the issue of computation in real-time).

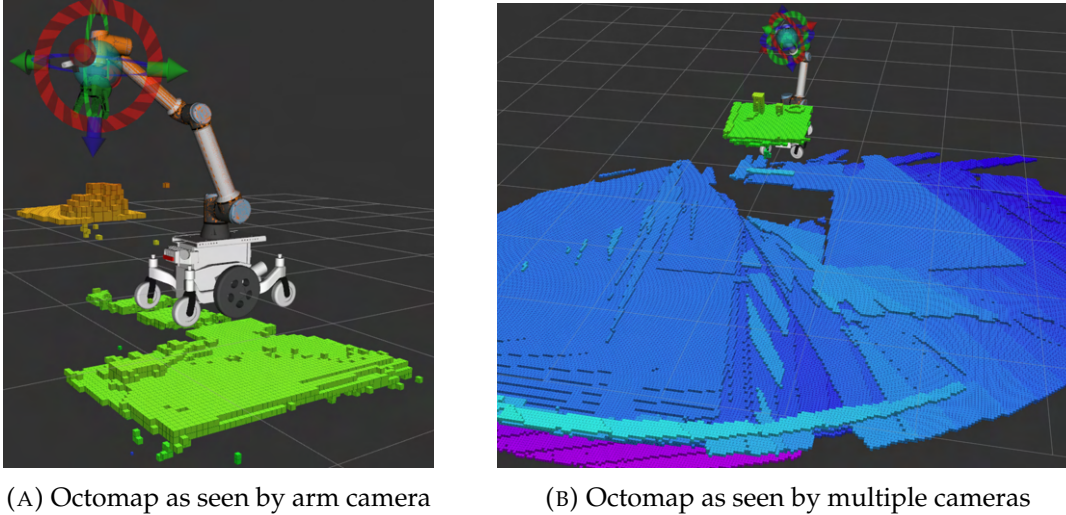
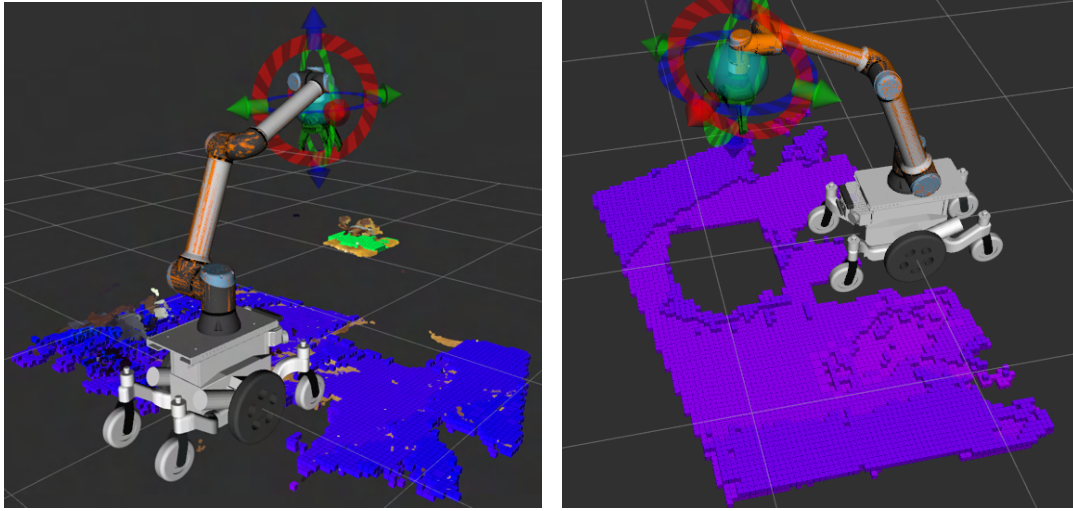


FIGURE 7.10: Examples of Octomap-based collision world scenes from different sources of cameras in Amiga

Combing things – all planes are obstacles: While "all-are-obstacles" approaches sound promising, there are certain cases where it does not apply. One sufficient example would be in Figure 7.9a from the last section. As this figure was adopted from the ZED 2 camera on the arm, it is apparent that the teaspoon was in the line of sight, which, obviously, was not an obstruction. To address this, we propose the "all-planes-are-obstacles" method, which is based on Amiga perception's plane detection node. Recall that we have built a node in C++ which remove the (one or more) planes contained in any object/region point cloud with at least 15 HZ. Now is the best time to evaluate this node's performance. Figure 7.11a illustrates an example outcome. The plane detection node extracts all the planes and publishes them in a point cloud, which is then caught up by the Octomap plugin. The dark purple colour represents the plane ground, the green colour represents the chair, and the objects on the ground (bags and chargers) and on the chair (a teddy bear) are not rendered as obstacles. Another point worth mentioning is that the first time we try to fit planes, we must configure and test various RANSAC parameters. However, since this node has been implemented in C++ for performance reasons, we will need to recompile/launch it every time we modify its source code. Due to the laborious nature of this process, we improved it by adding [dynamic_reconfigure, 2015] tools, which allow for testing different parameters interactively in a GUI. Now let's get back to Figure 7.9a's placing task example. In that case, only the chair and the

ground would be considered to be obstacles, and we can hence plan safely to anywhere on the frying pan.



(A) Octomap subscribing to plane-fitter point clouds (B) Octomap after manipulation regions are removed

FIGURE 7.11: Examples of Octomap-based obstacle avoidance combined with Amiga perception's plane fitting and dimension estimation

Combing things – identify manipulation regions: Depending on the context, the definition of obstacles naturally varies. When it comes to a robot cooking assistant, the objects/regions of manipulation sometimes should not be rendered as obstacles. The problem under our discussion context (Octomaps) essentially turns to be a point cloud segmentation problem, i.e, given a point cloud $PC(x, y, z)$, partition it into two-point clouds PC_{object} (or PC_{region}) and $PC_{obstacles}$. This task can be rather hard and is a vivid area of research today; for our purposes, we propose a simple primitive adding method relying on our initial approach¹⁵. We can manually add primitive shapes to the collisionObject in MveIt, and it will bypass collision checking with them during planning. Moreover, we can easily interact with Amiga perception to gain more autonomy. Figure 7.11b illustrates such an example. In this case, the dimension of the region point cloud¹⁶ is then estimated (Section 6.2.8) and fitted into a cylinder primitive.

Moreover, since our techniques so far are established based on the publishing/subscribing mechanisms, we need to develop methods to abort the communication in order to switch amongst them. For example, we can include a listener in the plane detection node such that if it catches a "stop" message, it will stop functioning. On the other hand, the Octomap stored in MoveIt's planning scenes can be removed by the rosservice /clear_octomap.

¹⁵See the primitive obstacles paragraph

¹⁶Recall from Amiga perception the region point cloud is based on object/region detection bounding box

7.2.10 Task-study : Stirring

Now that we have basically gone through the concepts inside our manipulation server. Let us demonstrate their power by applying them to specific kitchen tasks.

Consider a stirring task and its simplest case where we manually attach a known length of a chopstick to the gripper, and the task is to stir the content in a bowl. From the concepts developed in the last chapter, we can already retrieve the 3D coordinates of the points on the surface of a bowl (See Figure 6.5). We can use this information to estimate the bowl's centre and radius, move the virtual effector (the down-tip of the chopstick) there and generate an online trajectory, which can be a plane circle path that varies according to an input radius. A pseudo-code like:

```
# get current eef pose
x, y, z, rx, ry, rz, rw = getCurPoseProxy.call()
# get 3D surface points on the bowl
surface_points = rospy.wait_for_message('/amiga_perception
    /bowl/roi_3d_points', Vector3DArray)
# estimate the centre and radius from these points
centre, radius = estimateFromPoints(surface_points)
# generate a series of points for the chopstick end to go through
waypoints = [generateCircleTrajectory(radius-0.05,resolution=100)]
# assign the same orientation as the eef to these points
for waypoint in waypoints:
    waypoint.orientation = Quaternion(rx, ry, rz, rw)
# generate the centre of the bowl pose
target_pose = makePose(centre, [rx, ry, rz, rw])
```

should give us an initial destination pose in the camera's frame and 100 waypoints for the end of the chopstick to go through. We can then do:

```
# [0, 0, 0.4] = [tx, ty, tz] w.r.t eef frame
# (chopstick-end to eef offset may have tx and ty as well)
eef_pose = virtualEffectTransProxy.call(target_pose, [0, 0, 0.4])
eef_traj = virtualEffectTransProxy.call(waypoints, [0, 0, 0.4])
# regarding all planes as obstacles (because the chopstick is
# in the view, we cannot use 'allareobstacles')
obstacleAvoidProxy.call(type == 'allplanes')
planToTargetProxy.call(eef_pose, 'zed2_left_camera_link')
eef_traj_baselink = coordFrameChangeProxy.call(eef_traj,
    'zed2_left_camera_link', 'base_link')
planCartesianProxy.call(eef_traj_baselink)
```

such that a mixing task can be performed.

Now let us consider a more complicated case. Suppose now we want to grasp and then mix. The first thing to do is to grasp a tool such as a chopstick

or a spatula. Figure 7.12a illustrates this scene with an example of a spatula, although KitchenNet detects it as a chopstick. In order to perform subsequent stirring tasks, it is ideal for grasping the tip of the spatula (where there is a hole). However, let's assume Amiga grasping does not know how to do this and instead it uses some deep learning methods to do this as seen in Figure 7.12b to 7.12e (how we did this will be presented in the next chapter).

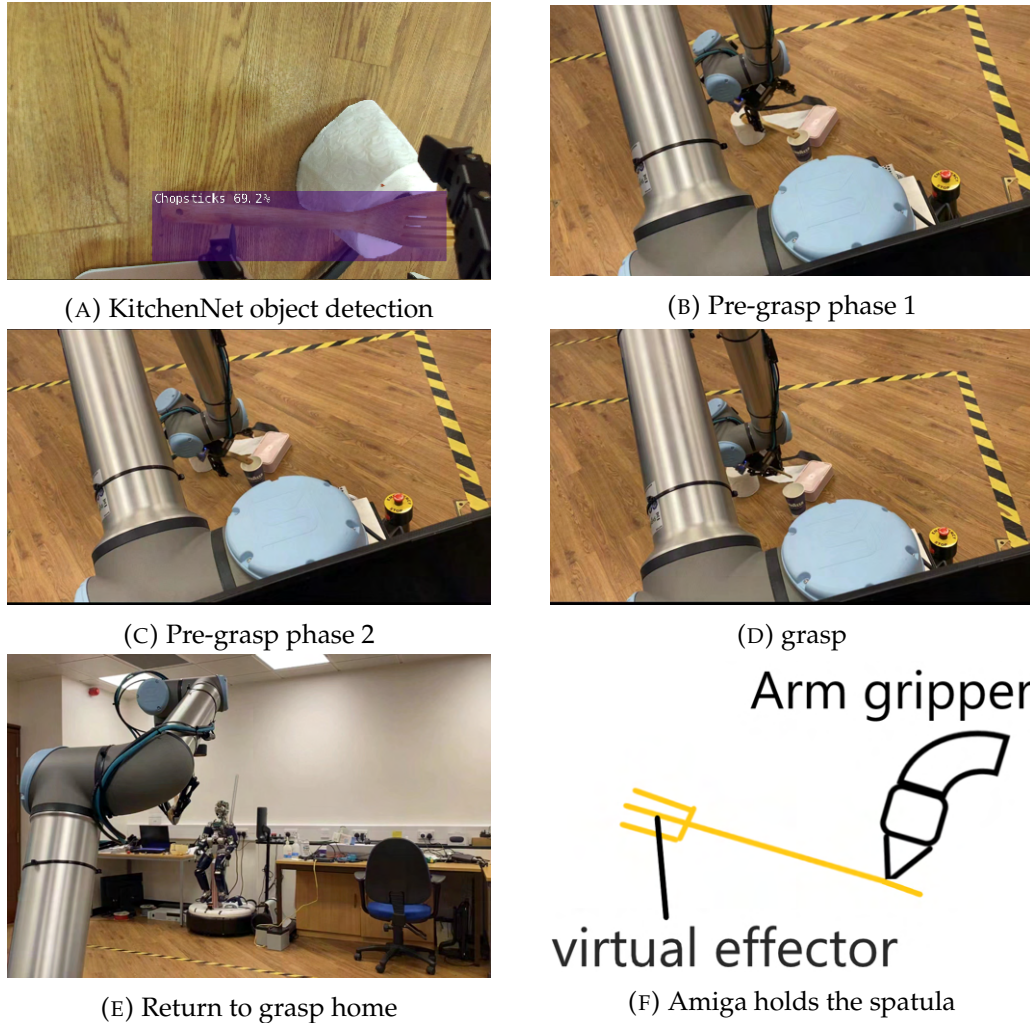


FIGURE 7.12: Amiga grasping a spatula

After a series of motions, we eventually landed back at the grasp home pose, as shown in Figure 7.12e. The image is unclear, so we draw it in Figure 7.12f. In this case, the point labelled in the figure is what we essentially want as the virtual effector. From our previous arguments, we would be able to achieve mixing provided we know the transformation between the virtual effector and the end effector. However, in this case, we will need to estimate this transformation ourselves. While this can be achieved via Amiga perception and some dimension estimation techniques, it introduces the first uncertainty. Furthermore, due to the high degrees of freedoms entailed by our deep grasping methods, the transformation between the two links may no longer be regarded as mere translations. Once the virtual effector is associated with a relative rotation, it can no longer be handled by our current

translator. Although we have proposed to use IKs to resolve this matter, it remains unclear to us the feasibility of translating a trajectory directly.

To conclude, for a combined grasp and mixing task, the uncertainties will be introduced by perception dimension estimation in conjunctions with the counterparts introduced by deep grasping methods (if used). There is a very high likelihood that the trajectories generated by perception will end up being executed at undesired positions.

7.2.11 Task-study : Pouring

In this section, we analyse a pouring task and investigate methods to attain this task.

While the simplest pouring can be achieved via rotating a joint angle, this method only works if the base_link has no relative rotation to the world frame and if the container is wide (for example, a bowl). For smaller containers and longer bottles, we will need to devise a better trajectory.

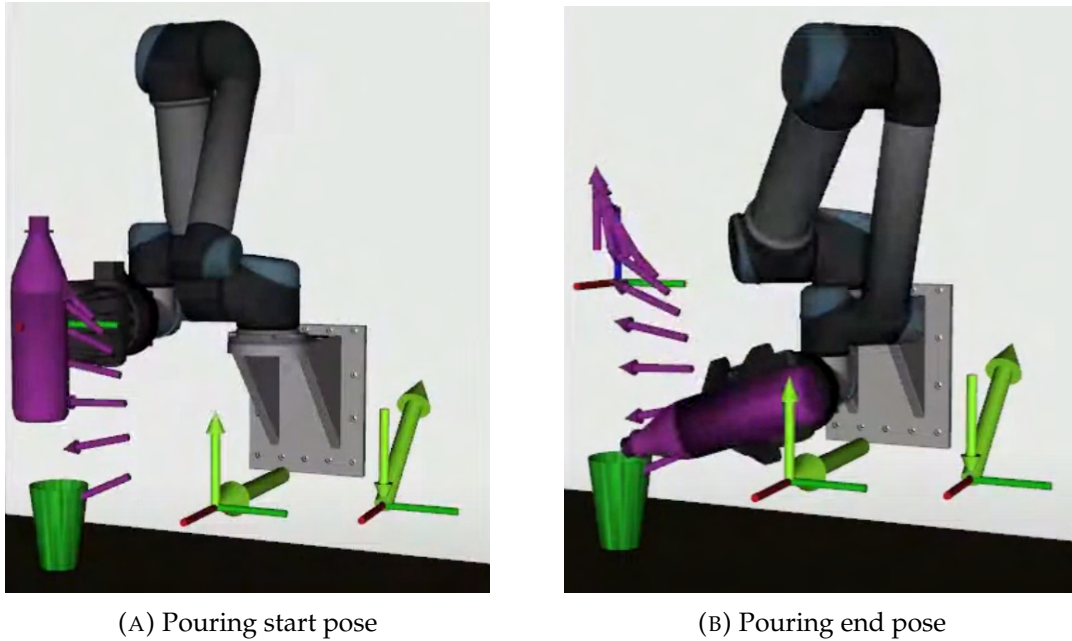


FIGURE 7.13: UR10e performing a pouring task [MoveIt_Task_Constructor, 2022]. The arm moves above the cup and then follows a pre-defined trajectory whose waypoints only concern changes in z coordinates and rotations around the x-axis (shown in red)

Figure 7.13 indicates a pouring task. After grasping the bottle, the arm moves to a temporary middle pose and then lands at the pose as indicated in Figure 7.13a. The pouring operation was then performed by following a pre-defined trajectory shown in Figure 7.13b, at the end of which the tip of the bottle collides with the cup, and liquids are flowing out. Essentially, the trajectories are devised such that the x and y coordinates of the tip of the bottle are not changed. Given the starting orientation of the bottle (which can be thought of as aligning up with the world's z-axis) and the final orientation

(which can be obtained via experiments or analytical methods), we have two methods to achieve the desired trajectories (setting the tip of the bottle as the virtual effector). The first method would be to put a constraint on the planner and request a plan. In this case, the constraints are, for the starting pose, p_1 and the end pose p_2 , we plan to from p_1 to p_2 such that only the z-coordinate is changed, and also the only relative rotation should be around the x-axis normal to the surface of the bottle, hence effectively 2 degrees of freedom. On the other hand, since only two variables exist, we could easily postulate the desired trajectories and sample uniformly some poses between the start and the end, setting them as waypoints for the virtual effector (tip of the bottle) to traverse, hence completing the tasks (as shown in the figure by the purple axes). The starting pose above the cup can be inferred by the cup's object point cloud published by Amiga Perception.

7.2.12 Generic Task-solving

Now that we have examined two cases let us take a small step towards generalisation.

During the early stage of this project, we were given the option to program the arm via a build-in teach-and-repeat paradigm method, where the robot is shown the actions by a human demonstrator and a series of waypoints and control signals are stored to repeat the actions on demand. Under this context, Amiga has no perception and could not perform helpful tasks. This is mainly because it can only generate offline trajectories for an effector to go through. By contrast, what Amiga perception essentially brought in is the allowance for online trajectory and action planning.

Let us think of some example tasks we achieved and how trajectories were generated:

1. **Placing:** the trajectories were generated by the planners given target pose
2. **Mixing:** the trajectories were generated by our functions which offers a variable adjustable to perception input
3. **Viewpoint adjusting:** essentially a combination between 1. and 2. Instead of solving for an optimal trajectory that goes through all the sampled poses, we use planners to reach each. As a result, sometimes there appear to be unreasonable paths due to the imperfection of IKs and planners

Let us exclude reinforcement learning and imitation learning method for now. A task can be thought of as consisting of many actions, which can be further divided into trajectories. Instead of optimising the task itself directly, which could be arbitrarily complex if not undefinable, we divided them into many trajectories. However, not all trajectories need to be optimised. For example, in the case of placing an apple on a plate, we could accept any trajectories provided the obstacle violation is not met. On the other hand, in

a pouring task, some trajectories may result in the spilling of the liquids and are hence not acceptable. In order to avoid this, one option we employed was to manually define an online trajectory with many desired waypoints, which accepts some variables from the perception side. In doing this, what essentially is achieved is passing the knowledge of a human to the robot via programming. Therefore, to solve a generic task, we could:

1. Add more constraints (joints, positions, waypoints) to the planners
2. Pass human knowledge by programming constrained online trajectories

From the perspective of our manipulation server so far, the latter method can be more effective.

7.3 Architecture Summary and Comments

Figure 7.14 indicates summaries Amiga manipulation, a hierarchical architecture designed for Amiga to attain Kitchen tasks. In Amiga manipulation, every node is essentially a service.

We started by integrating MoveIt to Amiga, built small build blocks as services, and gradually went upwards, creating useful services such as the virtual effector translator and the grasp executor. Moreover, relying on these blocks and incorporating vision topics from Amiga perception, we can handle more and more kitchen tasks.

As we build upward, the reliance on perception also goes up. While simple tasks like placing and stirring require a single message from Amiga perception, modules such as obstacle avoidance generally require continuous real-time callbacks on vision topics.

The concept of Amiga manipulation is more than just a server, but a general kitchen problem-solving strategy where a task can be thought of as consisting of smaller tasks. In addition, it is also a development strategy. MoveIt is an extensive framework with many features, but they do not necessarily all work on Amiga directly. We manually bridged such gaps so that we had more confidence in what could already be achieved and, depending on our need, we could integrate more features.

Further, since everything is a service, a human can easily interact with Amiga by calling the wrapped action services in both commanding or speeches.

In the next chapter, we will explore how the useful services implemented in this section would allow Amiga to perform complicated grasp operations.

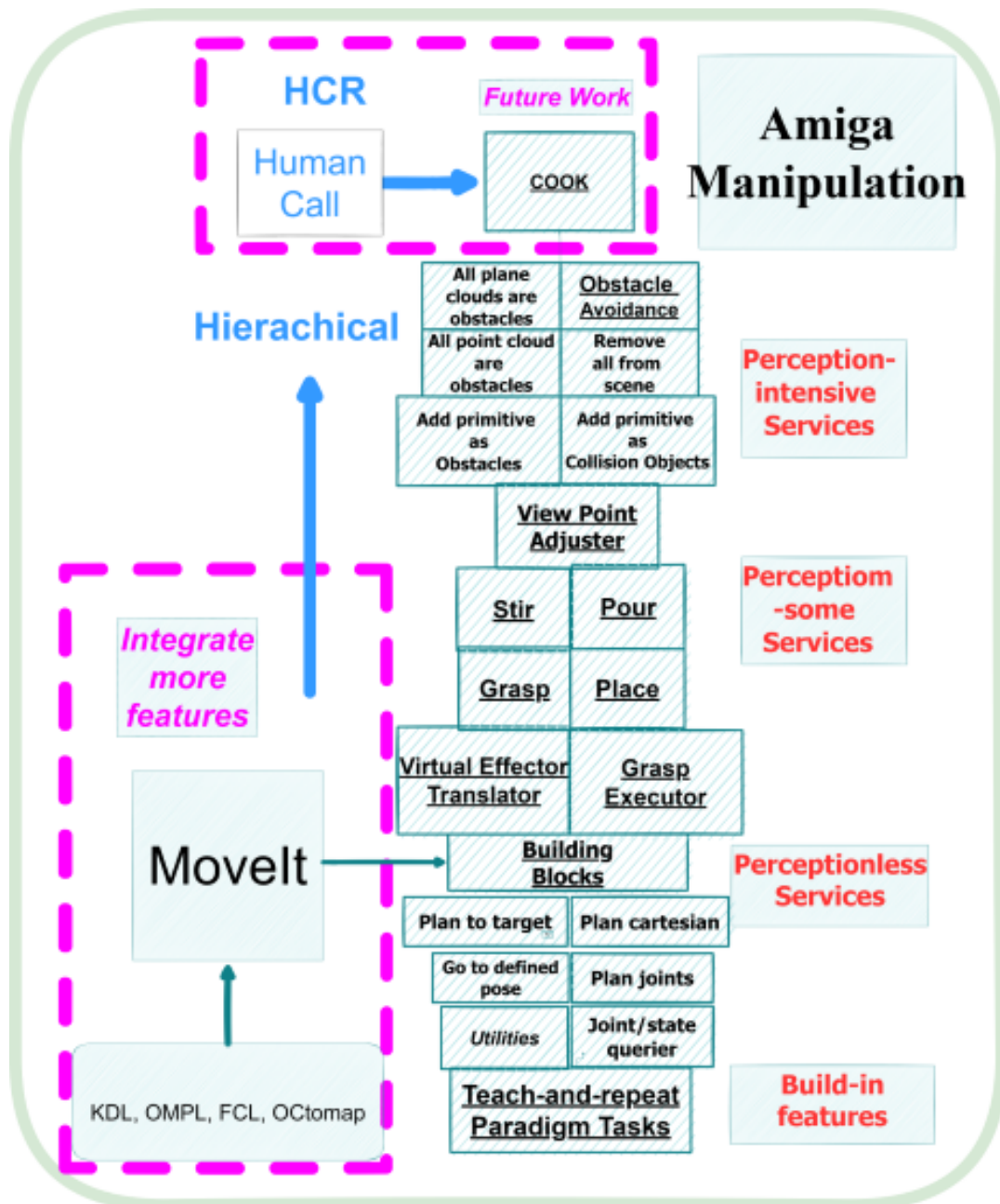


FIGURE 7.14: Amiga Manipulation in its entirety. The general principle is that complicated tasks can be assembled from smaller building blocks (hierarchy). From bottom to top, the level of interaction between perception grows. Pink dashes indicate future work

8 Amiga Grasping

8.1 Background and Module Design

Robotic Grasping is a pivotal aspect of manipulation. Despite coming naturally and intuitively to humans, robotic grasping is a complicated task, especially for unseen objects. Generally, a grasping process can be divided into three steps, two of which have already been addressed. The first step is computer vision, which locates the object and estimates its pose; the second step is grasp estimation, which infers an appropriate grasp pose; and the third step is motion planning, which calculates a feasible trajectory and passes it to the controllers. Still, a collection of one-step end-to-end methods based on reinforcement learning exist in the literature. For example, a network that perceives the spatial relationship between the gripper and objects in the scene, and hence learning hand-eye coordination, was proposed by Google in 2016 to servo the gripper for grasping directly [Levine et al., 2016]. As this project aspires to create a versatile modular structure, reinforcement learning methods for grasping will not be further investigated.

8.1.1 Grasping Pipeline

The crucial aspect of grasping an object is to decide the 6D gripper pose in the camera frame. Figure 8.1 demonstrates a typical grasp detection system pipeline. Many successful grasping algorithms from Amazon Picking Challenge [AmazonPickingChallenge, 2016] follow this scheme. They localise objects via image segmentation and estimate objects' poses via model matching. 3D point clouds derived from segmentation results are aligned with depth information and paired with existing models using *iterative closest point* methods in which the distance differences between two sets of point clouds are minimised. In addition, some methods utilised self-supervised learning to reduce dependence on data [Zeng et al., 2016], whereas others employed CAD models for 3D cloud point matching and achieved better results via *Monte Carlo tree search* [Mitash, Boularias, and Bekris, 2017].

The same scheme can be applied to this project as well. So far, object localisation can be resolved by object detection techniques as discussed in Section 6.1.1. Given that motion planning (Section 7.1) is a core component of the project already, it seems that one would need two more modules: pose estimation and grasp estimation. Given pose estimation (Section 6.1.2) results, grasp poses can be computed manually and stored in a database.

However, specifically with respect to this project, designing the grasping module based on this scheme entails several concerns:

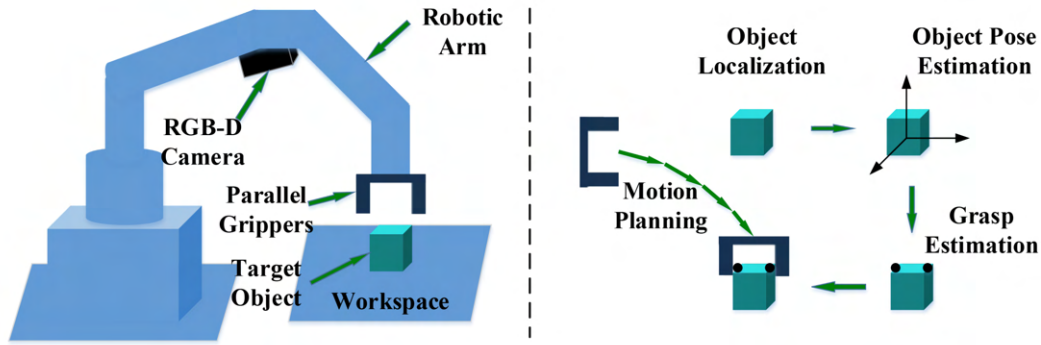


FIGURE 8.1: Grasp detection system pipeline [Du et al., 2020]. We have already implemented object localisation and motion planning in the previous chapters

- **Unification Concern:** As already mentioned in Section 6.1.2's Remark 9, the computer vision module should unit the object detection and pose estimation such that both function on the same set of objects. This has been decided to be left as future work due to the lack of a mature kitchen object detector.
- **Dependency Concern:** Our general principle of design is that modules should be decoupled as much as possible. It will be in violation of this principle if the grasping module relies entirely on the perception module's performance.
- **Generality Concern:** It would be ideal if unseen objects could also be grasped, rather than just objects whose models are already registered within the database).

Despite the discussion above, not all grasping operations are obligated to be based on object localisation and pose estimation – some methods can grasp irrespective of whether the computer vision system is functioning correctly. These methods usually come in an end-to-end fashion via deep learning and are reckoned more suitable for the grasping module's design.

Remark

Using pose estimation in conjunction with existing model matching to obtain a grasp pose has the primary benefit of being more explicable and reliable. The ability to trust that Amiga will be able to grasp some objects with almost perfect precision is always a comforting thing to know. This feature can be maintained simplified, bypassing pose estimation – retrieving the 3D position from objection detection&depth map and appending it with a pre-determined 3D orientation. In addition, incorporating end-to-end grasp estimation methods ensures greater flexibility.

The estimation of the grasping poses can be categorised into 2D planar grasp and 6DoF grasp [Du, 2021]. The grasping module will cover both by replicating two architectures, Dexnet [Mahler et al., 2016][Mahler et al.,

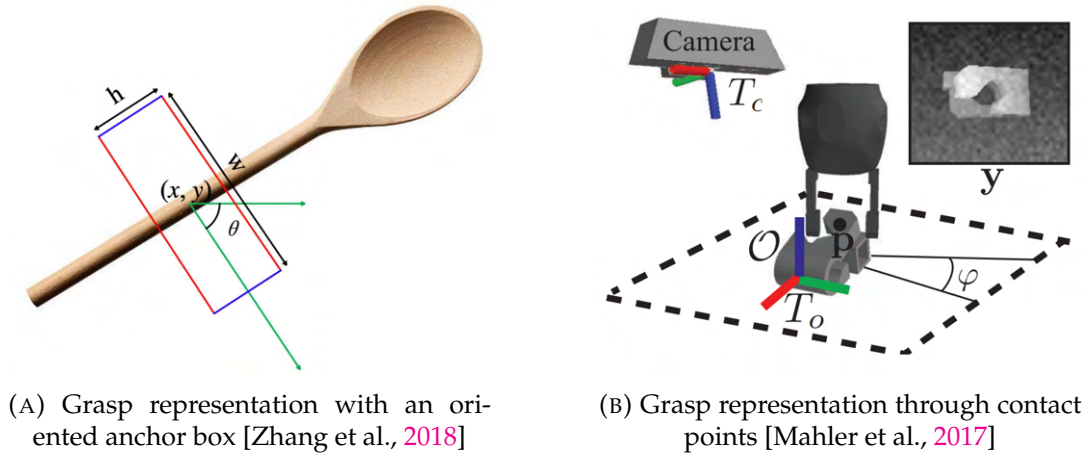


FIGURE 8.2: Two common planar grasp representations (oriented anchor boxes and antipodal contact points)

2017] and Grasp Pose Detection (GPD) [Pas et al., 2017]. The two methods have been selected primarily on the basis of their representation of the fields and open-source codebases. The following subsections will provide more detailed information with respect to our choices.

8.1.2 Planar Grasps and Dexnet

2D planar grasps assume that the objects are positioned on a plane workspace. Similar to a mobile robot case, these constraints reduce the problem from 6D to 3D consisting of two translations, i.e. x, y and one orientation about the z -axis, i.e. yaw. Constrained 2D grasps can be represented by an oriented anchor box [Jiang, Moseson, and Saxena, 2011]. As shown in Figure 8.2a, this grasp configuration, g_{oa} , is initially proposed in [Caldera, Rassau, and Chai, 2018] and consists of five dimensions:

$$g_{oa} = (x, y, \theta, h, w)$$

where (x, y) represents the centre of the rectangle, θ the orientation of the rectangle in relation to the horizontal axis, and (h, w) the height and width.

Similar to an objection detection task and with the only difference being an added term for gripper orientation [Redmon and Angelova, 2014], problems of this setup can be attained by CNNs by treating it as a mixture of classification and regression problems, which proved successful despite clustered scenes as in [Zhou et al., 2018].

Another popular representation for planar grasps is indicated in 8.2b, which is based on evaluation of contact points. The parallel-jaw grasp in 3D space consists of four dimensions:

$$g_{cp} = (\mathbf{p}, \varphi) \in \mathbb{R}^3 \times \mathcal{S}^1$$

where $\mathbf{p} = (x, y, z)$ specifies a centre with respect to the camera and $\varphi \in \mathcal{S}^1$ defines an angle in the table plane. The idea behind it is that parallel-jaw

grips are meant to place the contact points on opposite sides of the object; this is also known as an *antipodal grasp*.

Since deep learning methods are considered for implementing the planar grasping module, one must consider the available Dataset. Presumably, models trained on a larger dataset are more likely to generalise well. The largest public 2D planar grasp Dataset to date is the Dex-Net 2.0 Dataset [Mahler et al., 2017], which contains over 150 classes of objects, 6.7M depth images and grasp labels. This is much larger than its equivalences, such as the Cornell Grasping Dataset [Redmon and Angelova, 2014] and the CMU Dataset [Pinto and Gupta, 2015], which only contain 885 and 50567 RGB-D images respectively. Therefore, Dex-Net and contact-point-based methods will be adopted as a solution for planar grasping.

A significant discrepancy between orientated-anchor methods and contact-point-based methods lies in the usage of CNNs. In contrast to the former, which uses CNNs directly to determine grasp pose, the latter utilises CNNs to evaluate the quality of grasps sampled through an array of geometrical algorithms. For instance, the Grasp Quality CNN (GQCNN) model (Figure 8.3) which is trained on the Dex-Net 2.0 Dataset and contains more than 180 million parameters, inputs a grasp candidate $u = (i, j, z, \phi)$ and outputs a probability of grasp success $Q_\theta \in [0, 1]$.

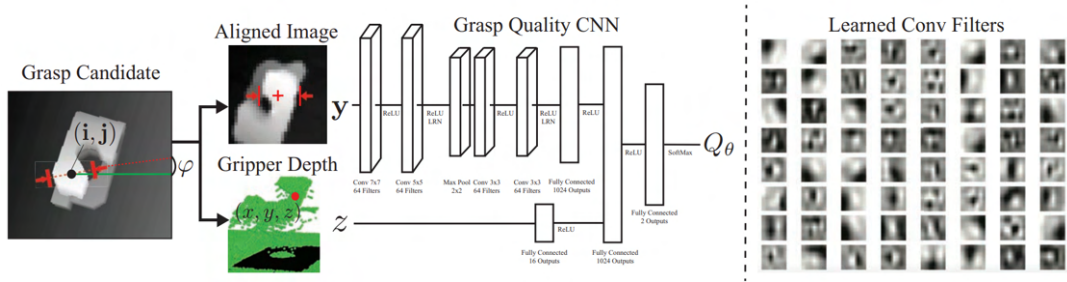


FIGURE 8.3: GQCNN architecture. Planar grasp candidates are acquired from a depth image and transformed to align the image with the grasp centre pixel (i, j) and orientation ϕ . Gradients of oriented images derived from the learned filters are calculated at varying scales, which helps in the determination of contact normals and collisions between the gripper and the object [Mahler et al., 2017].

Practically, training a network with more than 6.9 million images would require extensive engineering work; therefore, one may consider directly employing a trained model from Dex-Net. For this module, we employ the GQCNN-4.0-PJ pre-trained model, which has been presented lately in Dex-Net 4.0 [Mahler et al., 2019] and trained on images of objects in clutter. This specific choice does entail several concerns, which are summarised as below:

Remark

We note that GQ-CNN models can be susceptible to the parameters utilised during data generation, which might differ significantly from our project's setup. Specifically, these include:

- 1) **Gripper Geometry:** the trained model used assumes an ABB YuMi Parallel Jaw Gripper, whereas we use a Robotiq 3F gripper.
- 2) **Camera Intrinsics:** the trained model used assumes a PhotoNeo Phoxi S camera where we use an Intel Realsense / Zed2 camera.
- 3) **Distance to Workspace:** the trained model assumes 50-70 cm from camera to workspace during rendering, although this can be configured easily.

While the pre-trained model's performance is not guaranteed, relative evaluations will nevertheless be presented in the evaluation chapter.

8.1.3 6-DoF Grasps and GPD

A 6-DoF grasp, as the name suggests, enables the gripper to approach an object from a variety of angles. Unlike the planar grasps, 6-DoF grasps possess no posing constraints, and their representations vary according to the context. Figure 8.4 displays two examples.

In general, a grasp representation can be depicted similar to a pose representation:

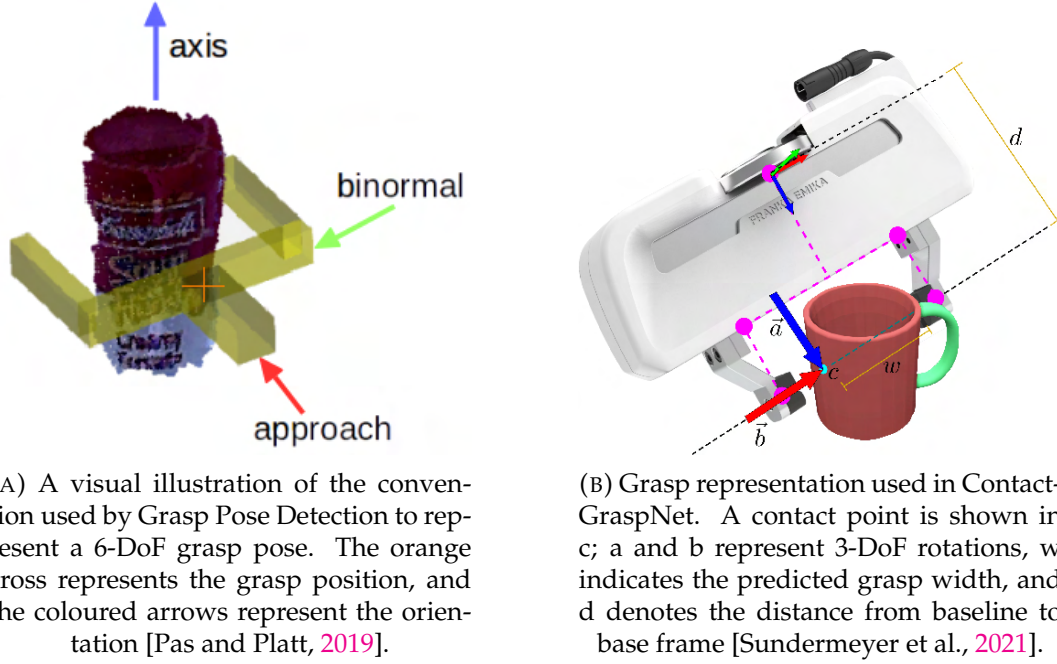
$$g_{6DoF} = (\mathbf{R}_g, t_g) \in \mathbb{SE}(3)^1$$

Similar to the planar grasp case, learning-based grasp estimation can be divided into sampling-scoring and end-to-end policy learning methods. Figure 8.4a shows the grasp pose detection method, which, similar to the Dex-Net case, uses CNNs to rank the quality of proposed grasps and return the most confident ones. By contrast, Figure 8.4b shows the GraspNet method, which generates 6-DoF grasp proposals with corresponding grasp widths directly from a raw depth image with optionally object masks. Both methods are mesh-free: no existing 3D mesh models are registered to match the point clouds. Moreover, it is worth mentioning that the point cloud is restricted to viewpoints. The process of inferring the whole shape is called *shape completion*. Both methods operate on partially available data.

To give an insight, for this project, we prefer the sampling-scoring methods to the direct regression methods mainly due to two reasons:

- According to [Mousavian, Eppner, and Fox, 2019], direct regression in high dimensional space such as $\mathbb{RE}(3)$ has proven to be difficult. This means these models might have generalisation issues with our setup, and integrating will bring about too much uncertainty.
- Learning a scorer rather than directly learning the policy allows for the enforcement of task-specific constraints without having to update the learned model [Mahler et al., 2017]. Re-training a deep learning grasping model is effortful work especially in our project where no baseline model exists yet; so we tend to avoid that in the initial implementation.

¹ $\mathbb{SE}(3) = \mathbb{R}^3 \times \mathbb{SO}(3)$ is the group of 3D poses



(A) A visual illustration of the convention used by Grasp Pose Detection to represent a 6-DoF grasp pose. The orange cross represents the grasp position, and the coloured arrows represent the orientation [Pas and Platt, 2019].

(B) Grasp representation used in Contact-GraspNet. A contact point is shown in c ; a and b represent 3-DoF rotations, w indicates the predicted grasp width, and d denotes the distance from baseline to base frame [Sundermeyer et al., 2021].

FIGURE 8.4: Two examples of a 6-DoF grasp representation

The Grasp Pose Detection method is selected to implement 6-DoF grasping modules. The high-level overview of the algorithm is described in Algorithm 2 below. One thing that worths emphasis is the hand config θ . It is assumed that a parallel gripper is used but we have three with our gripper – two fingers that oppose the third. While this can be modeled as a single contact surface by a virtual finger in between the two fingers on the same side, in the practical case, some of the feasible points in two-finger setup may be infeasible in three-finger case.

Algorithm 2 Grasp Pose Detection

Require: a viewpoint cloud C ; a region of interest R ; a hand config Θ

Ensure: $N \geq 0$

- | | |
|--|--------------------------------|
| 1: $C' = \text{Preproce}(C)$ | ▷ Cloud pre-processing |
| 2: $R = \text{CropROI}(C')$ | ▷ Crop a region of interest |
| 3: $S = \text{Sample}(C', R, \Theta, N)$ | ▷ Propose candidate grasps |
| 4: $I = \text{Encode}(S, C', R, \Theta)$ | ▷ Encode into CNN input |
| 5: $H = \text{Score}(I)$ | ▷ Predict candidate quality |
| 6: $g = \text{SelectGrasp}(S, H)$ | ▷ Return a set of 6-DoF grasps |
-

Remark

The reasoning behind choosing GPD in our design is summarised:

- **Sampling-Scoring:** as already discussed and as compared with direct regression methods, sampling-scoring methods like GPD possesses less uncertainty.
- **Segmentation-Free:** image segmentation is not implemented in the system. However, as a core feature, GPD requires no precise segmentation of objects and can fabricate hypotheses on any visible surface.

- **Miniature CNN:** GPD uses LeNet, an architecture that is relatively minor. Therefore, deployment can be simple, and inference speed can be assured despite high-dimensional inputs.

8.2 Implementation

8.2.1 Gripper Configuration

As a first step in working with grasping, we must configure Amiga's gripper. We set up ROS communication using [RobotIq_Driver, 2018]. In the `robotiq_3f_gripper_control` packages, the manufacturers provide a ROS firmware node written in C++. We wrapped it in ROS catkin and loaded it via `roslib`:

```
roslib.load_manifest('robotiq_3f_gripper_control')
```

Another package of interest is the `robotiq_3f_gripper_articulated_msgs`, which defines the message types for commuting between the gripper and ROS. In order to understand the mechanisms of the gripper and the meanings of each register specified in the package, one has to read the gripper user manual [RobotIq_User_Manual, 2019]. So far, control is possible via ROS messages where different registers map to different gripper statuses. However, this is not elegant because of a low level of abstraction. Instead, we could implement it as a ROS server. The list of features we support now is summarised in table 8.1 below (can be easily expanded in the future):

Service Function	Input	Output
Initialise the gripper	Empty	Empty
Reset the gripper	Empty	Empty
Open the gripper	Empty	Empty
Close the gripper	Empty	Empty
Change to basic mode	Empty	Empty
Change to pinch mode	Empty	Empty
Change to wide mode	Empty	Empty
Change to scissor mode	Empty	Empty
Set gripper force	Int (0 - 255)	Empty
Set gripper velocity	Int (0 - 255)	Empty
Check object contact	Empty	Bool (0/1)

TABLE 8.1: Available gripper services right now

Note that the object contact relies on a built-in object detection feature that other complex pattern recognition algorithms can replace in the future. Also,

the gripper force ranges from 15 to 60 N and the speed from 22 to 110 mm/s, which are mapped to integers 0 - 255.

As an example, to close the gripper, we can now command:

```
rosservice call /amiga_gripper/close_gripper
```

8.2.2 MoveIt Grasps

The initial implementation was based on [MoveIt_Grasps, 2020], a C++ grasping package from MoveIt.

The package features three main components:

- Grasp Generator: contain algorithm samples grasp poses based on the kinematics of the end effector and the shape of the object in order to optimise them using geometric scoring functions.
- Grasp Filter: assesses the feasibility of grasp potentials by searching for IK solutions to ensure they can be reached
- Grasp Planner: generates Cartesian approach, lift and retreat motions which comprise an entire grasp motion.

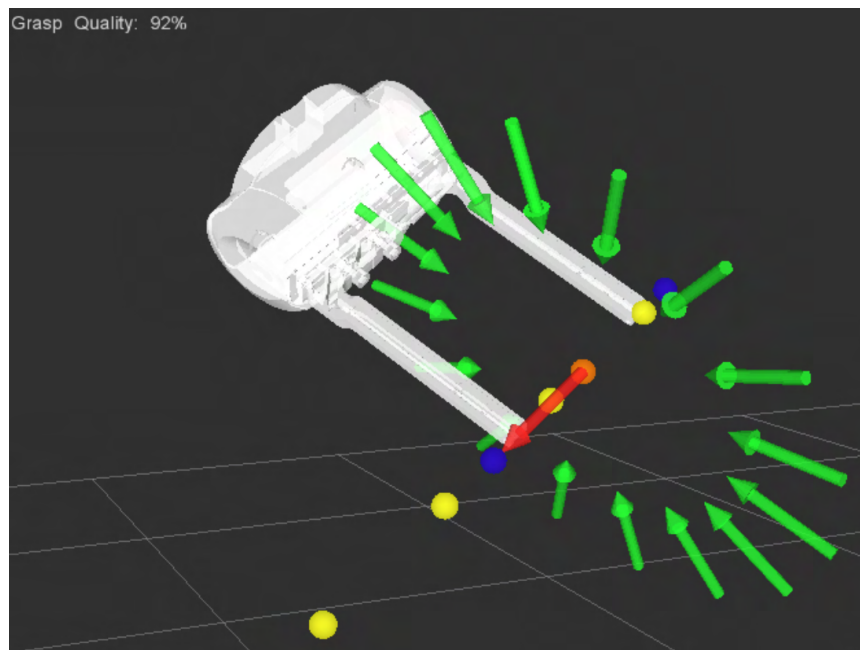


FIGURE 8.5: Pose-based grasp generator [MoveIt_Grasps, 2020]

We explored the code base of the package and understood its workflow. In short, MoveIt grasp first generates a grasp pose for a primitive-shape object and then proposes many potential grasp approaches and directions as in 8.5. These grasps are then filtered by removing kinematically infeasible

grasps using threaded² IK solvers. The concept within the grasp planner is similar to our discussions in Section 7.2.6, so we could use our own instead. The grasp generator and the grasp filter, on the other hand, inspire us to produce an embryonic form for Amiga's grasping.

Similar to the manipulator server, we propose the grasping server. In addition to accommodating gripper functionalities as mentioned in the previous section, this server essentially does two things: grasp generation and grasp filtering. The latter is adopted from MoveIt grasp directly.

Relying on MoveIt's grasp generator and Amiga manipulation³, we can now grasp primitive cylinder objects. However, can we do better?

8.2.3 3-DOF Grasping

In these section, we introduce and implement the concepts of 3-DoF grasping. The idea is as follows:

1. Retrieve from Amiga perception a ROI 3D coordinate
2. Append this 3D coordinate with a pre-defined orientation to make it a pose. This pose should now be in the camera's frame

We begin by defining two gripper orientations:

$$\begin{bmatrix} q_{down}^1 \\ q_{down}^2 \\ q_{down}^3 \\ q_{down}^4 \end{bmatrix} = \begin{bmatrix} -0.8903691114115917 \\ -0.45523902110162956 \\ 0.00015021799829083296 \\ 0.0005065028287560422 \end{bmatrix} \quad (8.1)$$

$$\begin{bmatrix} q_{front}^1 \\ q_{front}^2 \\ q_{front}^3 \\ q_{front}^4 \end{bmatrix} = \begin{bmatrix} 0.49956006551396753 \\ 0.5004395699227152 \\ 0.49991382218159247 \\ 0.5000861407708072 \end{bmatrix} \quad (8.2)$$

where $[q_{down}^1 \dots q_{down}^4]$ corresponds to a vertically down gripper orientation and $[q_{front}^1 \dots q_{front}^4]$ corresponds to an into-the-paper front orientation.

We can then pair up the first orientation with a top-down viewed position and start grasping. Pseudocode like

```
gripperInitProxy.call()
goToGraspHomePoseProxy.call()
x, y, z = rospy.wait_for_message('/amiga_perception
```

²Multiple instances at once

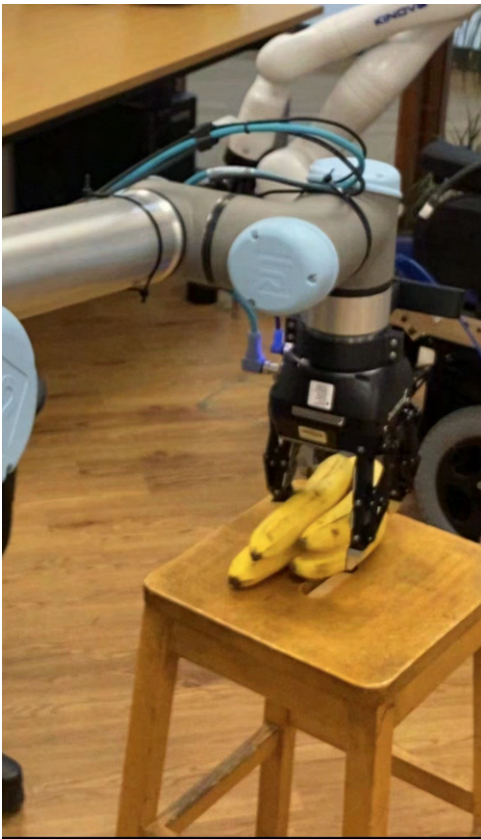
³Recall that all we need is to send the grasp pose to Amiga manipulation's grasp executor, close the gripper and then go to home pose (for example)

```

    /banana/roi_3d_point_bbox_centre', Vector3)
Pose = makePose(x, y, z, q_down)
# recall section 8.2.6
graspExecutorProxy.call(Pose)
gripperCloseProxy.call()

```

should go to grasp home pose first, then wait for a ROI 3D coordinate from the centre of the banana bounding box and plan a grasp. Moreover, the gripper needs to be initialised every time Amiga re-launches.



(A) Top-down banana



(B) Top-down orange

FIGURE 8.6: Examples of 3-DoF grasps

Figure 8.6 illustrate two examples. On the other hand, q_{front} is proposed to grasp objects such as bottles. For example, think of a pouring task – a top-down grasp, despite its success in grasping, is clearly unable to manage the subsequent pouring task.

Amiga perception provides considerable freedoms when it comes to choosing the 3D coordinates. We would like to use the "no free lunch" principle as a general rule – it all depends on the use cases. Suppose, for example, that we want Amiga to grasp a bottle of coke while in the same scene, a glass of the bottle is present, and the KitchenNet detects both as bottles. In that case, we would probably benefit from Amiga perception's secondary image processing features. Glass bottles are essentially texture-free; hence interest

point detection would be a good solution. The 3D ROI position can then be estimated from the average of the ROI 3D coordinates of interest.

Towards viewpoint-free 3-DoF grasping Furthermore, we might have already realised that our pre-defined gripper orientations align with Amiga's end-effector's orientation during scene viewing. For example, at the grasp home pose, the end-effector's orientation aligns with the grippers' counterpart (we already addressed this in the last chapter). Therefore, we can eliminate the pre-defined orientation phase and generalise 3-DoF grasping to a viewpoint-free version. Suppose Amiga is now in any unknown pose observing the banana; we do:

```
# get eef pose; eef orientation = gripper orientation
x, y, z, rx, ry, rz, rw = getCurPoseProxy.call()
x', y', z' = rospy.wait_for_message('/amiga_perception
    /banana/roi_3d_point', Vector3)
Pose = makePose(x', y', z', rx, ry, rz, rw)
graspExecutorProxy.call(Pose)
gripperCloseProxy.call()
```

such that Amiga would attempt a grasp at that particular viewpoint. The word "any" implies greater freedom but less control. To exert control, we could define poses for Amiga and then drive it to those poses before grasping in order to gain an understanding of what can be expected. The grasping home pose and the grasping front pose are the two useful poses we think of, but we could always define more. These are the essential concepts for our 3-DoF grasping feature. The viewpoint adjuster (Section 7.2.7) can be used to test whether generic 3-DoF algorithms (essentially a perception problem) are robust enough to viewpoints. Nevertheless, we would argue that these methods would not generalise further beyond objects of a specific shape. For example, a spherical apple, for which every view is projected to a circle and hence the centre of which, appended with current gripper orientation, should theoretically achieve a good result.

Let us follow the grasp pose convention in Figure 8.4a to define a gripper pose. In the case of 3-DoF, the banana in Figure 8.6a can no longer be grasped if it has a relative rotation around the approach axis. To address this issue, and according to this chapter's background, we adopted more sophisticated algorithms based on GQ-CNNs.

8.2.4 Replicating GPD

Recall from the background that we selected Grasp Pose Detection to implement our 6-DoF grasping module. To do this, one has to replicate GPD first.

The codebase in GPD is developed in C++. One of the major issues we encountered was segmentation faults during compiling; this was ultimately resolved by removing a flag for compiler optimisation.

The following configuration files were required:

1. A Caffe LeNet model weight (we used 15channels)
2. An `eigen_params.cfg` specifying parameters defined in the paper (for our interest, we should modify the gripper geometry and the number of samples)

The CNN is optimised via [openvino, 2022], and some of the library conflicts were resolved via virtual environments. The author provided a ROS wrapper; although it could not be run directly, we managed to fix the issues having made some effort. Moreover, GPD required point cloud format as XYZRGBA, and this could be resolved by setting the alpha channel to 1.

Figure 8.7 illustrates GPD's inference on a single object. These data were collected by saving as a pcd format and visualised with Open3D. The raw camera point clouds input was too large (more than 800,000 points) and could not be handled by GPD's preprocessing; hence we employed Amiga perception's plane removing node to remove the table and the ground. Therefore, the input to GPD in this example was just a bowl and a cup. As a result, the inference for these two object meshes only took 0.1s. Further, in Figure 8.7, the blue grippers indicate sampled grasp poses, and the red&yellow grippers indicate the grasp with the highest score from the QC-LeNet. After our modifications with gripper geometry, the results looked very satisfactory.

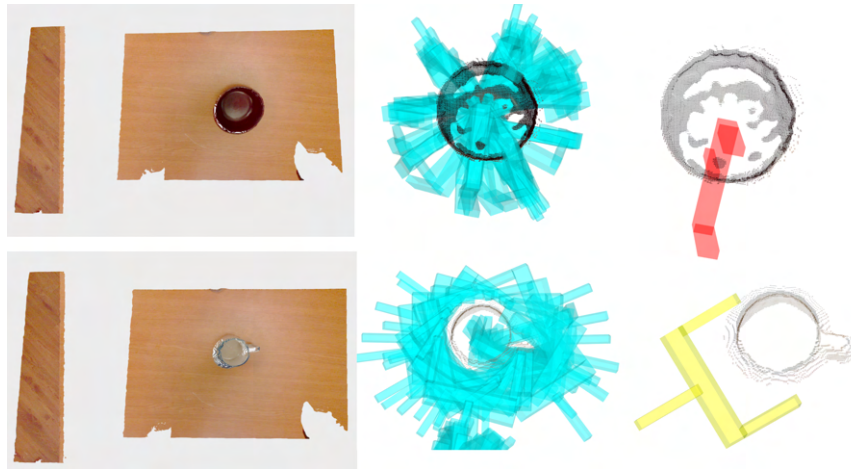


FIGURE 8.7: GPD single object inference. Left: raw point cloud. Middle: proposed grasp candidates. Right: final selected grasp. The gap in the bottom right corner of the table in the raw point cloud is because of gripper occlusion

Further, Figure 8.8 visualises the top 5 most scored grasps by GPD on dense clustered kitchen region point clouds pre-processed by Amiga perception's region detection in conjunction with plane removal. The figures are displayed in ascending order of crowdedness. In Figure 8.8a, three grasps favoured a pear, and one green grasp selected a sponge; the grasps were quite reasonable. In Figure 8.8b, more objects were put in, and the grasps picked different objects such as a bottle of water, a coffee cup and a tissue box with some tissue out. The cluster-level increased even further as in 8.8c,

but GPD still returned very reliable outcomes on acceptable grasps. However, a human would render the green grasp infeasible due to its contact with the tissue box. The level of the crowdedness was maximised in 8.8d, but GPD still gave good grasps, with four favouring the tropical juice and one favouring a spoon.

As the size of point clouds increases, the time taken for inference also grows. As far as we can tell, all inferences were performed within 2 seconds with a sampling number of 1000, and one will need to reduce the sampling number in trade for time performance for larger inputs.

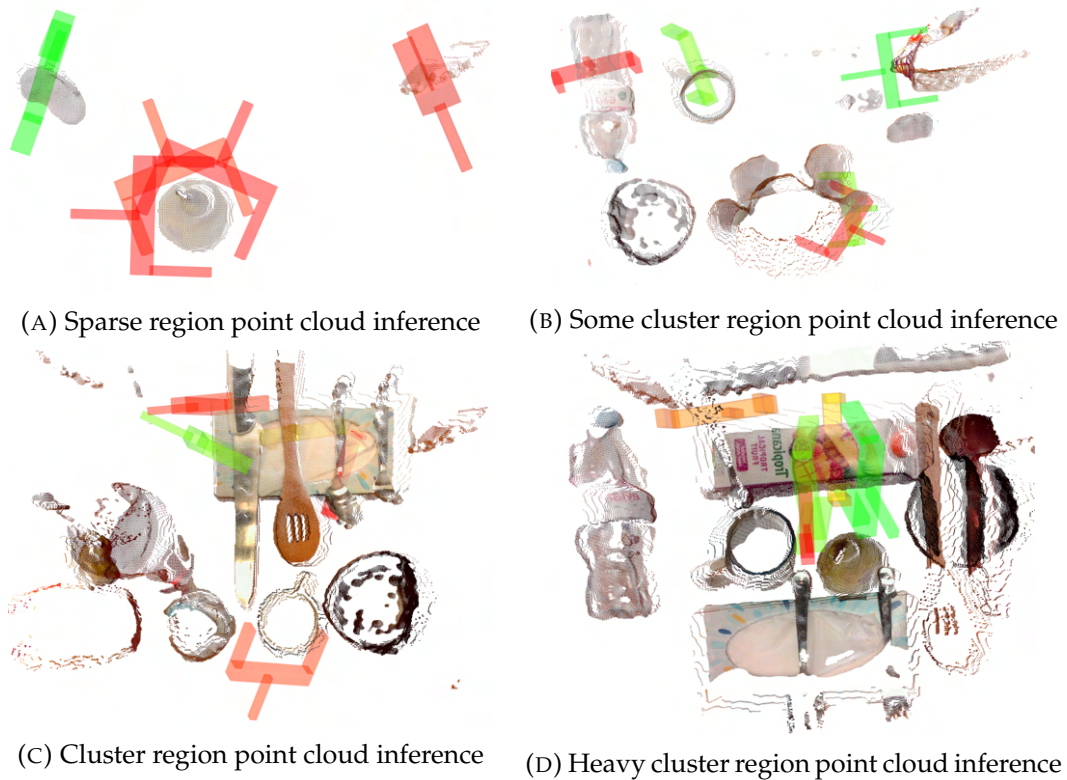


FIGURE 8.8: GPD inference on 4 different real-world dense region point clouds from Amiga's view in a kitchen context

8.2.5 Replicating Dex-Net

Recall from the background that we selected Dex-Net to implement our planar grasping module. Now let us replicate Dex-Net. This process was very effortful because the codebases were subject to inconsistencies, outdated dependencies and numerous library conflicts. In order to use Dex-Net, one has to install multiple `berkeleyautomation`'s dependencies, including `gqcn`, `autolab_core`, `perception`, and `visualisation`.

Our implementation was inspired by [Deep_Grasp_Demo, 2020], which uses MoveIt Task Constructor [MoveIt_Task_Constructor, 2022] to implement deep grasps in simulation with a panda arm. Since we were to implement in reality with our manipulation server and Amiga, the discrepancies

were very high. However, we would reuse the srv definitions and the action lib provided in the package. The following configuration files/folders were required:

1. A camera_calib_intr file for the camera (ZED 2 in our case)
2. A dexnet_model_config (GQCNN-4.0-PJ in our case; parameters tuned to resemble robotiq 3f gripper as much as possible)
3. A script to run dexnet_policy (Python version ≤ 3.6)
4. A path to store binary grasp candidate samples and RGBD images

Figure 8.9 provides an overview of our integration to Dexnet in a simplified manner. Suppose we have launched the query server, passing knowledge regarding 1, 2, and the location of 3. In short, a query node call_backs a real-time RGB image and an aligned depth image and saves them to a folder. It also sends a ROS action lib request to the query server. The query server accepts the request and launch a python 3.6 script, passing knowledge via subprocess.communication (an one-way communication method). The script then checks the folder to retrieve the images, converting them into Berkeley Perception's RgbdImage format and query policies from Dex-Net, which accepts a RgbdImage, an optional segmask (in our case, none) and a camera intrinsic file. Dex-Net then proposes grasp samples via geometric methods, queries the quality values for each of them using QC-CNN, and returns the best grasp to the script, which then dumps to the folder (we used pickle) for the query server to de-serialise. The query server finally looked up the result and returned an action result indicating the success of the inference. The 6-DoF grasp poses can be retrieved from the action feedback topics.

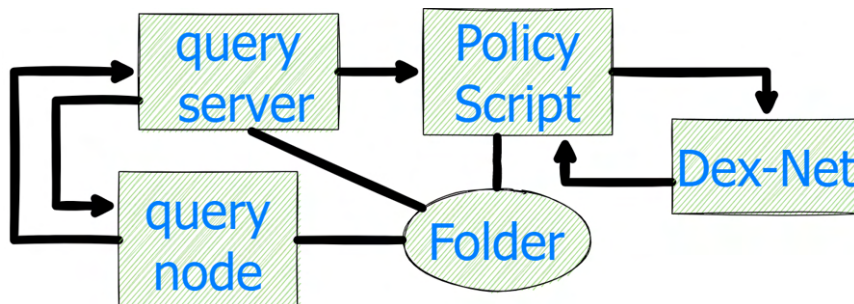


FIGURE 8.9: Integration of Dexnet to our system. A query server launches a script running Dex-Net and saves the results to a folder. The server looks up and publish query feedback

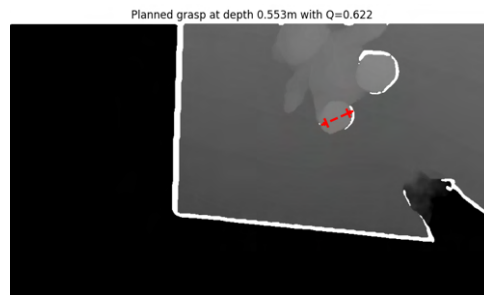
A few facts to note:

- **Script:** The use of the policy script is essential because Dex-Net relies on TensorFlow 1 [Abadi et al., 2015], which no longer supports python versions > 3.6 . We were using 3.8, and the rest of the system depends on 3.8, so we ended up installing two versions of python (virtual environments did not work). We specified using python 3.6 to run the policy scripts via a bash command.

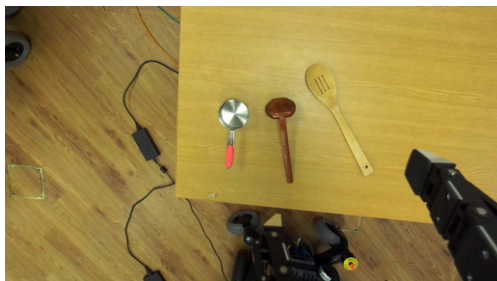
- **NPY format:** In the beginning, the images were saved as CVBridge encoded image formats and passed for queries. Later on, we discovered that using NumPy files would be four times as faster. Conversions from ROS to NumPy can be achieved via a package called `ros_numpy`. As a result, the system responds to the query node within 1-2 seconds.
- **Folder:** The use of the folder is for two purposes: first, to serve as the communication bridge between the query server and the policy script; second, to allow us to collect data. For this course, we also put efforts into developing utility scripts to save and render the depth images published by the cameras correctly (L515 and ZED 2 have different encodings, we adopted `rqt_image`'s source code for rendering). Certainly, the connection between the query node and the folder can be further optimised so that any devices in Amiga network can query the generated grasp poses in the future.



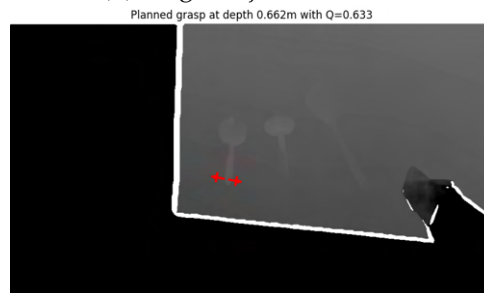
(A) Single object raw rgb image



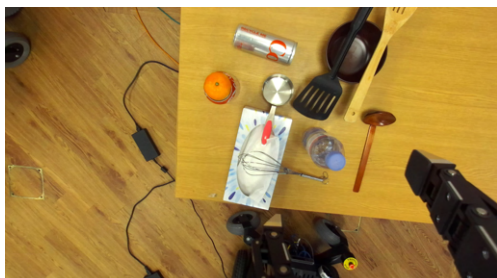
(B) Single object inference



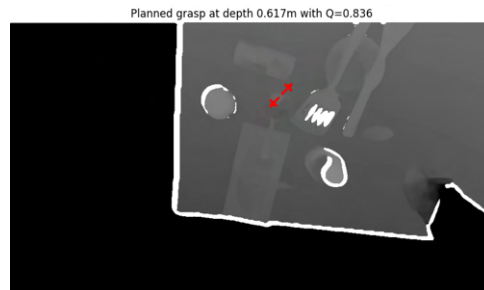
(C) Some cluster raw rgb image



(D) Some cluster inference



(E) Heavy cluster raw rgb image



(F) Heavy Cluster inference

FIGURE 8.10: Dex-Net inference on real-world RGB-D data

Figure 8.10 illustrates three inference examples from Dex-Net on our real-world RGB-D images. All images are taken in the grasp home pose, and

the input depth images have been pre-processed using the region detection algorithm we proposed in Section 6.9 and specifically in Figure 6.9a.

Figure 8.10a and 8.10b suggest an example inference on a teddy bear. In this case, Dex-Net provides a reasonable grasp pose with a quality score of 0.622 despite that the teddy bear is not presented in the QC-CNN model database. Objects that are more related to kitchen contexts are tested in Figure 8.10c and 8.10d. Dex-Net predicts a fairly reasonable grasp pose on the teaspoon in this scene with some clusters. Figure 8.10e and 8.10f show example inferences for heavily clustered common kitchen objects. Dex-Net predicts a very reasonable grasp pose on a teaspoon with a high score of 0.836. Dex-net predicts very well on teaspoons because, upon checking, the model database contains meshes similar to them.

8.2.6 6-DOF Grasping

In this section, we apply GPD and DexNet we replicated in Section 8.2.4 into reality with Amiga.

The grasp pose generated from GPD is stored in the `zed2_left_camera_frame`, and this knowledge is passed via the camera info. Consequently, the 3D position of the grasp pose will be stored in the camera frame. However, the 3D orientation in this pose is not. GPD has no knowledge of Amiga's gripper's coordinate frame system. The 3D orientation from the grasp pose is essentially a rotation with respect to its pre-defined gripper pose (Figure 8.4a from Background 8.1.3). After performing some rotations in our mind (Note that the order is yaw, pitch, then roll), we concluded that the relative rotation from Amiga's gripper to GPD's gripper definition is:

$$(\phi, \theta, \psi) = \left(-\frac{5\pi}{4}, -\frac{\pi}{2}, -\frac{3\pi}{2} \right) \quad (8.3)$$

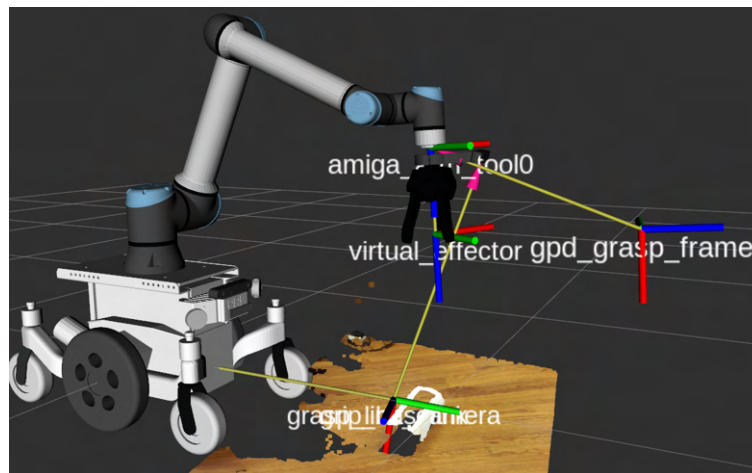


FIGURE 8.11: Various frames in a 6-DoF grasping via GPD on a kitchen roll. The GPD frame is visualised by postulating its transform with respect to `amiga_arm_tool0`

Figure 8.11 shows an example case where we used GPD to grasp a roll of kitchen paper. GPD's gripper definition frame is visualised using tf⁴ and all frames are with respect to a fixed coordinate system. According to the gripper pose definition, it is apparent that, with reference to the gpd gripper, the grasp pose will concern a relative rotation of approximately 90° around the x-axis, which is reasonable in order to grasp the kitchen roll.

Now let $\{A\}\mathbf{R}_{\{B\}}$ defines the rotation from frame $\{A\}$ to $\{B\}$, we now have $\{eef\}\mathbf{R}_{\{gpd\}}$, $\{gpd\}\mathbf{R}_{\{target\}}$ and hence we can compute $\{eef\}\mathbf{R}_{\{target\}}$ by multiplying them together. However, what we essentially want is a rotation in the camera's frame, $\{camera\}\mathbf{R}_{\{target\}}$; to obtain this, we will need to multiply $\{camera\}\mathbf{R}_{\{eef\}}$ with $\{eef\}\mathbf{R}_{\{target\}}$.

More over, recall in Preliminary 2.2, the rotation matrix $\{camera\}\mathbf{R}_{\{target\}}$ from Euler has the form:

$$\{camera\}\mathbf{R}_{\{target\}} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Continuing to that, in order to make a target pose, we will need the $(\phi_{target}, \theta_{target}, \psi_{target})$ in the camera frame, which can be obtained via:

$$\phi_{target} = \tan^{-1} \left(\frac{r_{21}}{r_{11}} \right) = \text{atan2}(r_{21}, r_{11}) \quad (8.4)$$

$$\theta_{target} = -\sin^{-1}(r_{31}) = -\text{asin}(r_{31}) \quad (8.5)$$

$$\psi_{target} = \tan^{-1} \left(\frac{r_{32}}{r_{33}} \right) = \text{atan2}(r_{32}, r_{33}) \quad (8.6)$$

While our hand-written implementation was subject to unknown errors, we discovered that tf already provides useful functions such as `quaternion_from_matrix` and `quaternion_matrix` for our purposes.

Combining with the 3D grasp coordinate in the camera frame, we have successfully obtained the 'actual' grasp target pose for Amiga in the camera's frame. We can send it to the grasp executor⁵:

```
pose_base_link = transPose(
    target_pose, from="zed2_left_camera_link", to="base_link")
graspExecutorProxy.call(pose_base_link)
gripperCloseProxy.call()
```

although many experiments suggested, we may want to use MoveIt's IK-based grasp filter beforehand. In Grasp pose detection's original paper, the authors mentioned that they used IKFast (for IK solutions) and OpenRAVE

⁴Gently ignore the translation as it is only for better visualisation

⁵Recall that `trans_pose` was already implemented as in Appendix A.3.1

(for collision checking) to sample out kinematically infeasible grasps. Moreover, in order to perform cluster-removal tasks, they also devised a utility function that takes into account the grasp position's height, the required gripper width, the angle between the grasp approach vector and the robot base vertical axis, as well as the C-space distance the arm must travel to reach the grasp from a fixed, nominal configuration. Recall from Section 7.1.4 that we used KDL and FCL for IK and collision checking in Amiga manipulation and decided to switch to better solvers/checkers in case of ill-performance. Indeed, the ability to perform a 6-DoF grasp is probably one of the most influential metrics for evaluating their performances.

While we have some successful 6-DoF grasps with objects such as kitchen rolls and bowls, testing these grasps are very difficult due to a combination of problems related to both hardware and software, which we will state more clearly in the evaluation chapter. The most significant problem for our discussion was the non-logical paths executed to reach the desired target pose.

Figure 8.12 illustrates one of such examples. During an attempt to grasp the cloth, Amiga collided with one of the L515 cameras and entered the emergency stop state.

While one might argue that it might be because the transform of this L515 camera was not accurate (a slight offset did exist), but clearly, there could have existed better paths.

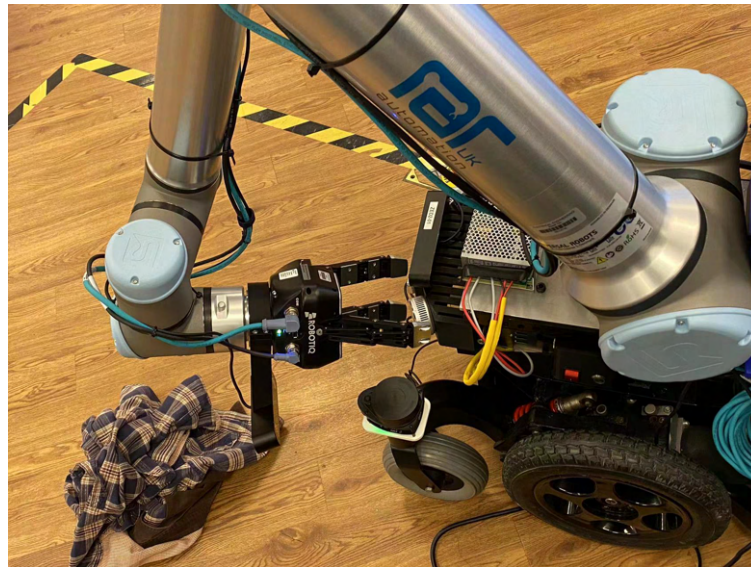


FIGURE 8.12: Bad path when trying to grasp a cloth via GPD. The gripper hit a camera during the path in reaching the desired target grasp pose

After many experiments, we speculate the sources of failures as follows (order matters):

1. Kinematic IK solvers and path planners
2. Collision checkers (ours has no continuous checking capability)

3. Algorithms itself (as postulated from Rviz & tf)
4. Base_link shifts as the arm moves (as a result of wheel slipping)
5. Point cloud registration errors

Consequently, it was realised that specifying only a target pose entails a high degree of freedom for the motion planning module as, essentially, only one waypoint is given. Furthermore, since the grasp pose is generated on-line and is 6-DoF, it is difficult to set more (meaningful) waypoints in advance.

8.2.7 5-DoF Grasping

Before addressing the bad-path problems due to high degrees of freedom, let us apply Dex-Net to reality first.

Figure 8.14 indicates a scenario where we employed Dex-Net in an attempt to perform a grasp in a region composed of a cup and a cantaloupe. Figure 8.13b depicts the corresponding Dex-Net inference result and the quality score (in this case, a full score of 1.0).

The output of Dex-Net 4.0 is also a 6-DoF grasp pose; therefore, we should refer to the original paper for its definition.

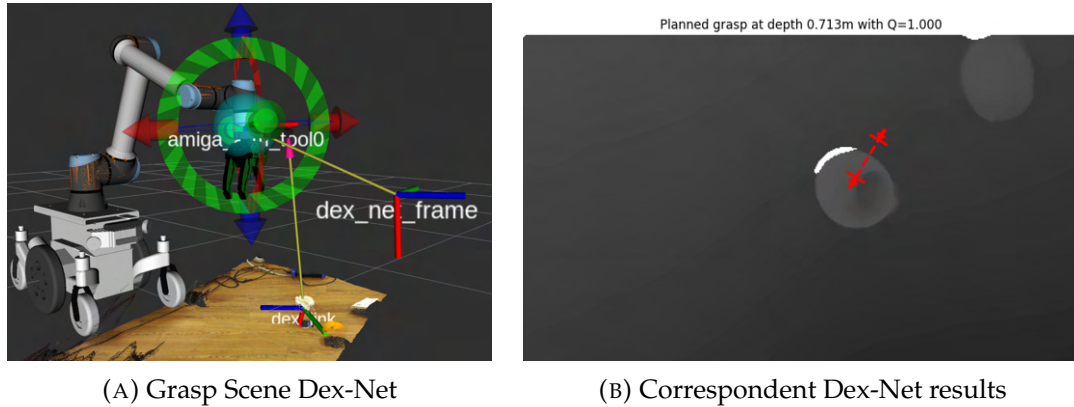


FIGURE 8.13: Real-world grasp inference via Dex-Net on a region of cluster. The result was shown in Figure 8.13b and the generated grasp pose is visualised in 8.13a

Figure 8.14a illustrates Dex-Net's grasp definition. The line connecting the two-finger contact points (shown in pink) defines the grasp axis. The y-axis of the gripper is parallel to the grasp axis, the x-axis is perpendicular to the table, and the z-axis is perpendicular to the grasp axis and parallel to the table. In this case, the grasp, similar to GPD's case, is defined separately from the object's pose for which there is only one defined component: the y-axis that connects the two contact points.

Again, we figured out the rotation from Amiga's gripper to the defined gripper as:

$$(\phi, \theta, \psi) = \left(\pi, -\frac{\pi}{2}, \frac{\pi}{4}\right) \quad (8.7)$$

with this knowledge, we visualised the Dex-Net grasp frame as visible from Figure 8.13a. However, unlike the GPD case, the grasp poses for Dex-Net is not fixed. Figure 8.14b illustrates the projection of our grasp reference frame onto the depth image inference results as in Figure 8.13b.

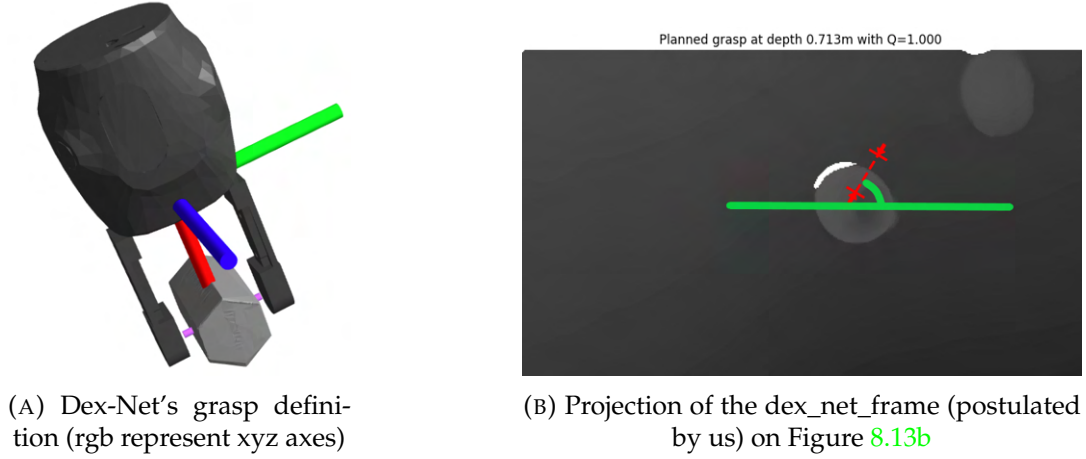


FIGURE 8.14: Dex-Net's grasp definition. Unlike GPD, the definition is not fixed. The only fixed component is the y-axis that connects the two contact points. Therefore, the rotation we figured out (Equation 8.7) is not general. Figure 8.14b shows the projection of the dex_net_frame according to this rotation

Now, given that the performance of 6-DoF grasping in the last section is restricted to the motion planning modules, it is natural for us to impose a constraint, hence reducing the degrees of freedom. We came up with two approaches:

1. **Waypoint Constraint (X)**: assign more waypoints from the current pose to the target pose. However, as already discussed, it is difficult to sample or assign meaningful waypoints in a 6-DoF's case
2. **Viewpoint Constraint (✓)**: recall from this chapter's background that Dex-Net is mostly employed for planar grasps. A planar grasp can be view-dependent as opposed to 6-DoF grasping, which is essentially view-free. However, our top-down predefined grasp home pose already eliminates the variation of viewpoints; therefore, we could activate this pose constraint and generate 5-DoF planar grasp pose

As a consequence of planar grasping, the grasp coordinate frame in Figure 8.14a needs to be rotated around the y-axis in order to point the x-axis downwards (but not perpendicular). This implies we will need to transform the raw grasp output from Dex-Net such that its x-axis is pointing towards the ground/table. Essentially, this entailed maximising the projection of the x-axis of the grasp pose onto the ground. The result is a 5-DoF grasp pose.

Following the same procedures described in the last section, we successfully executed the 5-DoF grasps. Until this stage, we could already grasp a variety of objects. However, although less frequently, the problems of unreasonable trajectories still occur, leaving room for more constraints.

8.2.8 4-DoF Grasping

In our case, the concept of 4-DoF grasping is essentially an engineering compromise. The circumstance was that we thought investigation in the optimisation sides of the motion planning module might exceed our time budgets and came to us that we could keep imposing more constraints on our generated grasp poses.

Given that we have had the transformation in Equation 8.7, we could compute an angle as visualised in Figure 8.14b and regard it as the only reference source for a grasp pose. This process essentially reduces the 3D orientation to 1D, leaving only a relative rotation around the z-axis in the world frame. In other words, the grasp coordinate frame in Figure 8.14a is rotated such that the x-axis now points perpendicular to the ground (recall in the 5-DoF's case, it is only downward).

The primary benefit of this approach is that we could be released from the limitations of path planning and kinematics feasibility. Planning and reaching these 4-DoF grasp poses (as they concern only one rotation) are much more manageable than their 5-DoF and 6-DoF counterparts. To some extent, this reduces our grasping system's dependency on Amiga manipulation.

As grasping was typically a task with relatively high error tolerances, we still managed to grasp objects such as coffee cups and teaspoons. Similar analogies can be applied to GPD as well. Moreover, exhaustive testing of this grasp approach must be performed to conclude its applicability and potential. Due to the limited time we had, this will leave as future work.

8.2.9 Object-aware Deep Grasp

Many inputs to the deep learning grasping methods we introduced in the last few sections are object regions generated via region detection techniques, whether in a cluster scene or not. We would like to refer these grasps as object-agnostic grasping – Amiga essentially does not know the type of objects it intends to grasp (only how to grasp).

Recall Amiga perception architecture as in Figure 6.13. Relying upon object detection, we will be able to perform object-aware deep grasping.

To use GPD, we simply call back the 3D ground-free object point cloud and regard it as inputs. On the other hand, to use Dex-Net, we can call back object RGB-D images and apply the same procedure.

Therefore, we can wrap the whole grasping pipeline in high-level services such that we can achieve something like:

```
# grasp an apple in 3-DoF, use current gripper pose as on
# orientation and centre of bounding box as position
goToGraspHomePoseProxy.call()
Amiga.grasp(object="apple", type="3-DoF", method="centre of bbox")

# grasp a beer in 3-DoF, use current gripper pose as
# orientation and average of SIFT as position
```

```

goToGraspFrontPoseProxy.call()
Amiga.grasp(object="beer", type="3-DoF", method="average of SIFT")

# grasp a teaspoon in 4-DoF, use dim-reduced grasp poses
# from Dex-Net based on perpendicular assumptions
goToGraspHomePoseProxy.call()
Amiga.grasp(object="teaspoon", type="4-DoF", method="Dex-Net")

# grasp a bowl in 6-DoF from any current pose, use dim-kept
# grasp poses from GPD directly
Amiga.grasp(object="bowl", type="6-DoF", method="GPD")

```

in accordance with the success of the detection of kitchen objects.

8.2.10 Towards Dexterous Grasping

As of now, our grasping strategy consists of only gripper closes, which can generally be applied to any parallel grippers. In addition, recall from the background that our deep grasping modules were built on a concept that a 3-Finger gripper could be regarded as a 2-Finger equivalence with a pseudo imaginary finger in the middle of the 2-Finger side. This inevitably introduced a 2F-to-3F gap, which we will leave for future research.

The point of discussion here is to explore the specifics of the Robotiq 3-F gripper in assisting Amiga's grasping, as otherwise, there is no reason to use it in the first place (although we already benefit from its build-in object detection for checking success of grasps).

Recall from Chapter 3 where we introduced Amiga. The Robotiq 3-F gripper contains multiple modes, such as basic, wide and scissor modes intended to grasp different types of objects, as summarised below:

1. Basic mode: the most versatile operation mode. It is best suited for objects that have one dimension longer than the other two
2. Wide mode: the mode optimised for grasping round or large objects
3. Pinch mode: the mode used for grasping small objects which require precise picking
4. Scissor mode: the mode used primarily for tiny objects. This mode is less powerful than the other three modes but is more precise

Now let us investigate the dexterous features entailed in the 3F-gripper by performing a grasp other than in basic mode. Figure 8.15a visualised the grasping virtual-effector in the scissor mode, as compared to our previous discussions where the grasp virtual-effector lies in the centre of the gripper.

Suppose now we want to perform a scissor grasp; if we just set the gripper mode to scissor, send the virtual scissor link to the desired target grasp pose and close the gripper, the object will not be grasped. This is because, in the

scissor mode, the two fingers on the same side essentially form a new gripper which entails a relative rotation of $\frac{\pi}{2}$ compared to the supposed one. Since this only consists of a single rotation, we can easily adjust the target pose and plan for a grasp; the result is depicted in 8.15b. Compared to the basic mode, the scissor mode could be more effective in grasping objects such as a bowl.

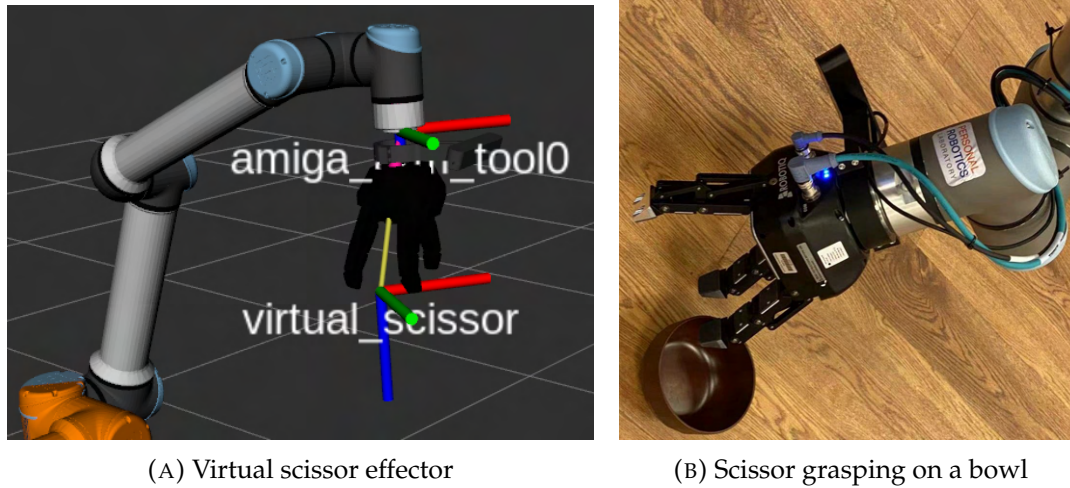


FIGURE 8.15: Illustration of the scissor grasping. Amiga regards the two fingers on the same side as a new scissor gripper and drives to the corresponding deep grasp pose

More thorough exploration of the 3F gripper's dexterity will be left for future investigation. An 11-degree-of-freedom gripper such as Robotiq-3F offers good prospects for dexterous grasping with Amiga.

8.2.11 Grasp-strategy Inference

This section is more of a theoretical exploration for future work. So far, we have proposed many grasping strategies. For example, since Dex-Net's GQ-CNN is trained from a top-down view, one might not want to apply it with a front view. Therefore during grasp planning, we would drive the arm to the grasp home pose and then perform either a 5-DoF or 4-DoF grasping. However, as the system now is a bit too general (aligning with the fact that Dex-Net was proposed for general pickings in the first place), exhaustive tests are needed to conclude both the accuracy and robustness when it comes to grasping a specific object following a particular strategy.

One way to improve trust in the system and the uncertainties introduced via deep grasping is resorting to historical data and establishing a mapping between objects and successful historical grasp strategies (which may also include different gripper modes), either view-free or view-dependent. The goal of having a trusted policy in grasping any objects in a kitchen is perhaps too ambitious. Therefore we may wish to begin by investigating and experimenting with the best strategies to grasp some common ones instead.

grasp objects such as an apple and a bottle. However, the types of objects graspable are still limited. This is improved by integrating deep sampling-scoring methods, Dex-Net and GPD. We began by directly adopting 6-DoF grasp poses from GPD but encountered kinematic and path planning problems and then attempted to lower the DoFs contained in a grasp pose. We reduced the dimension of the 6-DoF grasps from Dex-Net to 5-DoF and 4-DoF via activating the top-down viewing constraint and obtained satisfactory results. In general, as the degree of freedom inside a grasping pose decreases, the dependency on motion planning also decreases. Our current IK solvers and planners are not optimised enough for handling high DoFs robustly and will be replaced in the future.

Grasp Filtering: the poses (maybe multiple) from grasp generation can be filtered based on kinematical feasibility. This module is directly adopted from MoveIt's grasp package, although different IKs can be used. Moreover, the use of IK solvers to filter grasp poses and the grasp qualities judged by QC-CNNs face a trade-off. Slightly lower quality grasps can be more useful in practice if it is more kinematically feasible. We could investigate this further and assign a compromising weight to each factor.

Grasp Visualising: This step is to perform a manual check before executing. We may visualise the target grasp pose to see if it makes sense first and then visualise the planned trajectories to the poses in Rviz before executing.

Gripper Specifics: Different gripper parameters could be configured according to our gripper services. More features are left to be integrated from the Robotiq 3F gripper's user manual. Right now, we provide switching between different gripper modes as well as force and velocity control.

Grasp Planning: Depending on the gripper mode (i.e. basic vs scissor), send the modified grasp pose to Amiga manipulation's grasp executor for a complete grasp action.

Grasp Success Checking: This module now relies on a build-in object detection function from the Robotiq 3F gripper, but can be replaced with more complex pattern recognition techniques in the future.

Further work for this architecture can be plenty. The pink dashes indicate some possible directions.

1. **View-free testing:** the deep grasping modules' performances from a perspective other than grasp home/front pose is unknown and left to be explored in the future together with exhaustive testing.
2. **Object-pose-to-grasp-pose:** the grasp inference based on object pose meshes corresponds to Amiga perception's pose estimation future work. Inferring a grasp pose from known object meshes is desired if we wish

to grasp with a specific pose in order to perform certain subsequent tasks.

3. **Strategy inference:** providing a shortcut to achieve autonomy instead of explicit commanding by a human expert. This can be achieved via substantial experiments which thoroughly evaluate the reliability of grasping specific kitchen objects.

9 Amiga Navigation

9.1 Background and Module Design

Robotic navigation refers to the process of determining the robot's current position in its reference frame and planning a free path towards a goal location. This problem is similar to the manipulation problem but reduced to two dimensions as the wheels are bounded to the ground. A navigation problem consists of three sub-problems: *localisation*, *mapping* and *path planning*.

9.1.1 Odometry and Localisation

Localisation of a mobile robot is a challenging problem. As opposed to robot arms which only touch the air, the bases of mobile robots are constantly in contact with the ground, which is typically not flat.

Continued from Preliminary 2.7, the estimation of mobile robots' current location is known as *odometry*. Common methods include estimation of direction, velocity and time of travel, which are frequently accompanied by sensors such as wheel-encoders, IMUs and GPS. Uncontrollable factors such as wheel slips and rough surfaces accumulate over time, making localisation less accurate. Therefore, perfect motion integration from odometry is not possible. One solution is to turn to probabilistic approaches, which acknowledge the uncertainty and employ models to abstract useful information from data [Davision, 2022]. One such example is *sensor fusion*, the general process of combining data from many different sources into useful estimates. In modern navigation systems, sensor fusion techniques such as *kalman filter* are frequently used to fuse odometry information from different sources.

Moreover, another popular probabilistic localisation technique is *particle filtering*, which employs a set of particles $\{\mathbf{x}_i, w_i\}$ for $i = 1$ to N to represent the uncertainty of the robot's location. Each particle is a point estimate of the position associated with a normalised weight as in Equation 9.1.

$$\mathbf{x}_i = \begin{bmatrix} x_i \\ y_i \\ \psi_i \end{bmatrix}, \quad \sum_{i=1}^N w_i = 1 \quad (9.1)$$

As shown in Figure 9.1, the particle clouds represent uncertain robot states – the greater the number of particles in a region, the greater the likelihood that the robot is present.

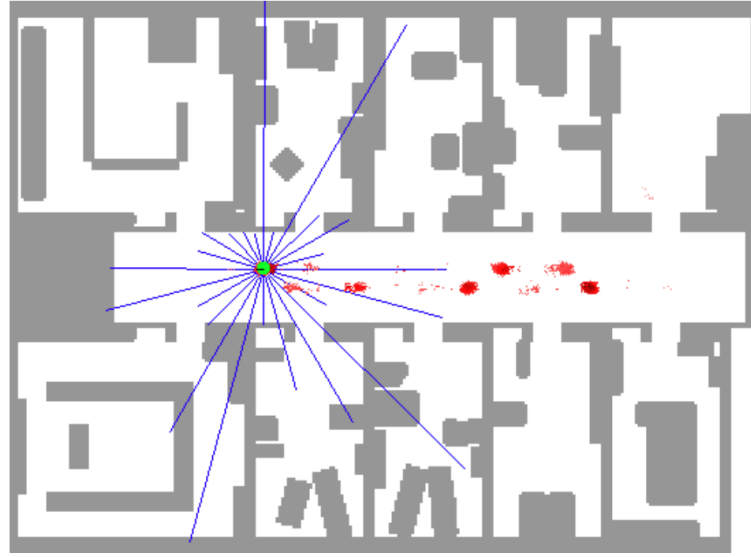


FIGURE 9.1: Monte Carlo Localisation (particle filter) using sonar sensors [Fox et al., 1999]

Every time the robot moves, a new sensor measurement is taken and the states of the particle are updated via *Bayes inference* as in Equation 9.2:

$$P(X|Z) = \frac{P(Z|X)P(X)}{P(Z)}, \quad w_i^{updated} = P(z|x_i) \times w_i \quad (9.2)$$

where z is a sensor measurement¹ and $P(z|x_i)$ is the likelihood of particle i . After the update, the weights of all particles are normalised and re-sampled such that, similar to a *genetic algorithm*, particles with larger weights are more likely to survive. The cycle repeats, and the particle clouds converge.

Remark

One of the core problems in implementing the localisation module is to determine the odometry source. As far as Amiga is concerned, several sources of odometry are available, such as the wheel encoder from the base, the camera, and the IMUs. Various strengths and weaknesses are inherent to each source. The wheel odometry, for example, is fast but not stable, and the visual odometry, by contrast, is precise but more computationally demanding.

9.1.2 SLAM and World Modelling

SLAM (simultaneous localisation and mapping) is a computational problem that involves the construction or updating of a map of an unknown environment while at the same time tracking where the robot is within the environment. SLAM remains a fundamental problem in mobile robotics in order to achieve true autonomous navigation.

¹The sensor is assumed to be of a particular distribution, typically Gaussian

The principle of SLAM is that we construct a map incrementally and localise the robot in relation to this map as it develops and is gradually refined. Most SLAM algorithms make maps of natural scene features [Davision, 2022]. For Laser and sonar sensors, these may be wall segments, planes and corners, whereas, for cameras, these may be salient point features, lines and textured surfaces. In order to ensure reliable matching, the features should be distinguishable and easily recognisable from different viewpoints.

One of the central assumptions in SLAM is that the world is static, under which the idea of probabilistic estimation extends to the map's features as well. On this occasion, the joint distribution over the states of both the robot and the mapped world is stored and updated. The *joint Gaussian distribution* is the most common and efficient way to represent the high-dimensional probability distributions we must propagate in SLAM. Updates can be accomplished using *Extended Kalman Filters*. The probability density function can be denoted by the state vector $\hat{\mathbf{x}}$ and covariance matrix $\mathbf{P}$ as in Equation 9.3, where $\hat{\mathbf{x}}$ contains the robot state (Equation 9.1) and all the feature states².

$$\hat{\mathbf{x}} = \begin{bmatrix} \hat{x}_v \\ \hat{y}_1 \\ \hat{y}_2 \\ \vdots \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} P_{xx} & P_{xy_1} & P_{xy_2} & \cdots \\ P_{y_1x} & P_{y_1y_1} & P_{y_1y_2} & \cdots \\ P_{y_2x} & P_{y_2y_1} & P_{y_2y_2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad (9.3)$$

As of today, numerous SLAM algorithms exist in the literature, such as LiDAR SLAM [Hess et al., 2016], Visual SLAM [Mur-Artal, Montiel, and Tardós, 2015], Dense SLAM [Newcombe et al., 2011], Graph-optimisation SLAM [Kümmerle et al., 2011] and more recently deep-learning-involved SLAM [Czarnowski et al., 2020].

Remark

SLAM is a vivid area of research. Existing SLAM algorithms can be categorised into 2D/3D representations and are typically performed with 2D/3D Lidar sensors and/or cameras. Recall from chapter 3 Amiga has several RGB-D cameras and two LiDARs available for navigation. In the absence of a baseline model, it is reasonable to adopt a strategy where popular ROS-integrated methods using these types of sensors are attempted initially, then gradually optimised or replaced with more advanced methods as required.

9.1.3 ROS Navigation Stack

Having established localisation and mapping, all that remains is path planning to guide us from the current position to the target. As such, this has been addressed in Section 7.1.3.

Putting together all the components for navigation is tedious work. Nevertheless, ROS provides a 2D navigation stack that uses odometry, sensor streams, and a goal pose to output safe velocity commands to a mobile base

² $\hat{x}_v$ is the robot state, (x, y, ψ) in 2D and $\hat{y}_i$ is feature state (X, Y) in 2D

[ROS_Navigation_Stack, 2022]. Figure 9.2 demonstrates the concept of the stack. The global cost_map is established from a map³ and the local cost map is updated on-line via the sensor sources whose transforms need to be provided. The recovery_behaviors are handled automatically. Upon inference, the server accepts a pose goal message, applies a global planner and a local planner (which also considers odometry), and outputs a series of twist messages to the base controller. The module also accepts an optional node *adaptive monte carlo localisation* (amcl) for probabilistic localisation.

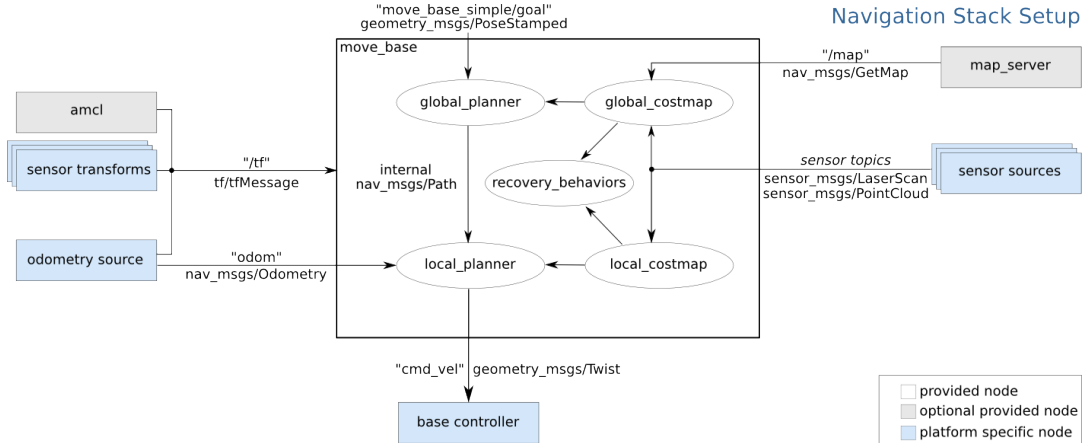


FIGURE 9.2: The mechanism within ROS_Navigation_Stack [ROS_Navigation_Stack, 2022]. Amiga's existing settings are pretty compatible with the system

Remark

In view of the nature of its navigation mechanics, differential drive, common odometry sources (wheel/vision odometry and IMUs) and shared sensor sources (lasers and RGB-D cameras), Amiga is highly compatible with the ROS_navigation_stack. This means, unlike the manipulation module, where many customary modifications are required, the navigation module can be prototyped quickly as a baseline model. Several advanced open-source navigation systems build upon the navigation stack; one famous example would be the Real-Time Appearance-Based Mapping [RTAB_Map, 2022], a RGB-D, Stereo and Lidar Graph-Based SLAM project based on an incremental appearance-based loop closure detector.

9.2 Implementation

9.2.1 Configuration and Investigation

In the same manner as with other modules, the first step is configuring the hardware and driver. Recall from Chapter 3 that Amiga has several cameras and two LiDARs for navigation. The LiDAR was set up via the [rplidar_ros,

³The map can be either off-line (pre-built) or on-line (SLAM)

2022] package. Despite the fact that the cameras have already been configured as part of previous modules' work, several of them will be used on this occasion. It is possible to use multiple camera streams as inputs to the visual odometry/SLAM algorithms, and hence two optimisation techniques were adopted:

1. set the `enable_sync` argument in the camera manager to true such that only messages that have the same timestamps (from different cameras) will be passed to the algorithms
2. employ [ros_nodelet, 2022] to simplify the process of running multiple cameras on a single device by eliminating the need to copy messages between processes (pass by reference instead)

Moreover, given that the lab has already provided the transforms of the cameras and RiDARs on the wheelchair base, the critical aspects of implementing the navigation module revolve around the odometry source, the sensor sources, and the establishment of the maps. For Amiga, we have investigated and explored several possible options regarding each aspect:

Odometry Sources

1. **Wheel odometry:** directly use the wheelchair base's wheel encoder under the topics of `arnie_controller/odometry`. In spite of the fact that this source is published frequently, errors can accumulate over time
2. **External visual and LiDAR odometry:** relay the RGB-D and LiDAR raw messages to the [rtabmap_odometry, 2021]'s odometry nodes, which include `rgbd_odometry`, `stereo_odometry` and `icp_odometry`. Due to the computationally intensive nature of this step, the published frequency is usually much slower than expected
3. **ZED 2 Visual odometry:** ZED 2 camera comes with a built-in position tracking feature as part of their SDK. It also supports optimised fusion with its IMU by setting the parameter `imu_fusion` to true. Compared to externally computed odometry, this internal approach is faster
4. **Kalman filtered odometry:** multiple sources of odometry can be fused into a single one in order to reduce uncertainty. The [ros_ekf, 2021] package implements an extended Kalman filter, a nonlinear version of the Kalman filter which linearises about an estimate of the current mean and covariance

Sensor sources

1. **RGB-D cameras:** the sensors generate depth images and the point clouds. Moreover, it is also possible to convert these messages to laser scans via [depthimage_to_laserscan, 2020] and [pointcloud_to_laserscan, 2015] packages. The latter is significantly more computation-intensive than the former. The camera-converted laser scans are usually not as accurate as of the equivalents from an actual LiDAR sensor

2. **LiDARs:** the sensors generate laser scan messages, which are a collection of single scan lines. The rplidars used in Amiga do not have built-in filters; efficient laser filtering can be achieved via [laser_filter, 2021] package to reduce redundant information and noises. Moreover, multiple single-plane lasers can be merged to generate a single virtual scan as in [Ballardini et al., 2014]

Mapping Techniques

1. **Gmapping:** a 2D laser-based SLAM is implemented in the [gmapping, 2019] package, which builds a 2-D occupancy grid map from the laser and pose data collected by the mobile robot
2. **RTAB-Map:** a RGB-D, Stereo and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector. It [RTAB_Map, 2022] can be used with a stereo camera (such as ZED 2) for 3D mapping and a 2D LiDAR for 2D mapping (such as Rplidar).

9.2.2 Gmapping and Amcl

Amiga is a giant robot, so it makes sense to avoid moving it around in the lab when we start. Instead, the simulation environment could be used to conduct experiments. Table 9.1 summarises the initial technique we adopted.

Odometry Source	Sensor Source	Mapping	Localisation
Wheel Encoder	LiDAR Only	2D Gmapping	AMCL

TABLE 9.1: Summary of the initial strategy we adopted for the navigation module in simulation

Figure 9.3 visualises the wheel odometry from the base for a short straight-line path. Messages are received and published quite frequently with acceptable accuracy, as indicated by the small yellow dots.

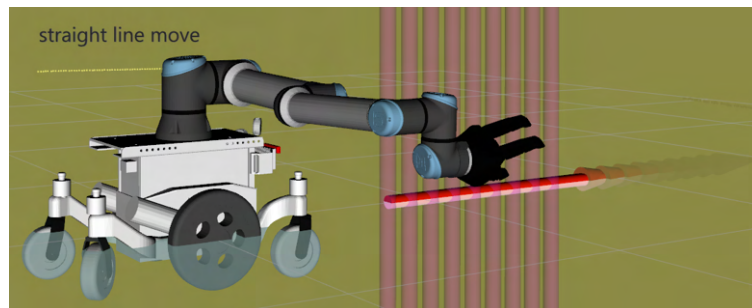


FIGURE 9.3: Visualization of odometry for straight-line movements. Red vertical cylinders indicate the different locations where Amiga has been at various times during the movement

The Gmapping node provides a transform from the map frame to the odom frame as part of localisation, and the odometry source publishes another virtual transform from the base_link to the odom frame. Afterwards, we can prepare and configure a yaml file for the node and launch the algorithm in the simulation environment as demonstrated in Figure 9.4.

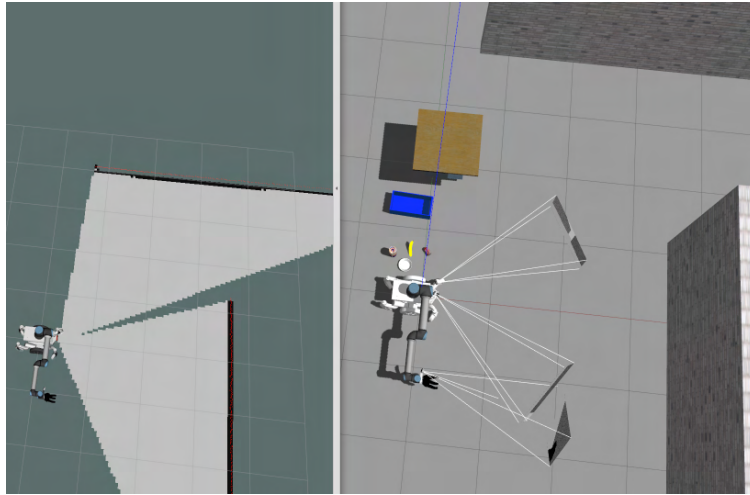


FIGURE 9.4: Online 2D grid map derived from Gmapping algorithms. On the left side of the figure, a laser scan of the simulated walls is being converted into grid lines indicating obstacles. A full map can be built after exploring the entire scene

A full grid map can be built after exploring the whole environment. According to our experiments, however, the odometry errors accumulated as we moved about Amiga, and the established map was no longer accurate. The noises can, however, be cleaned with a few external image editing tools.

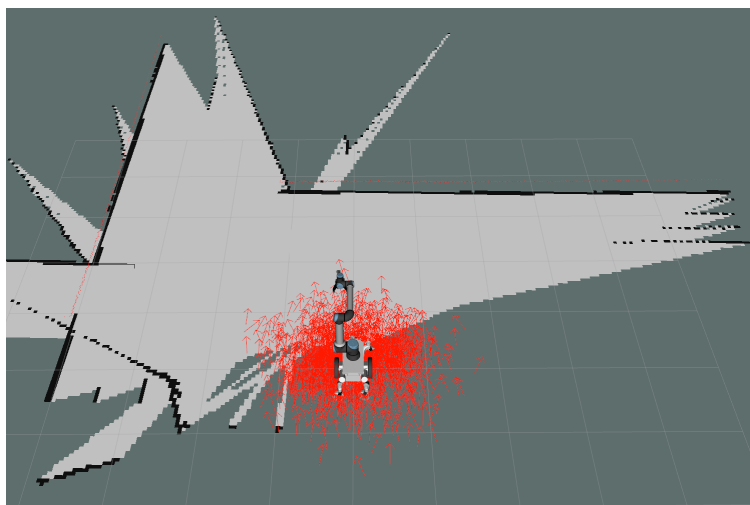
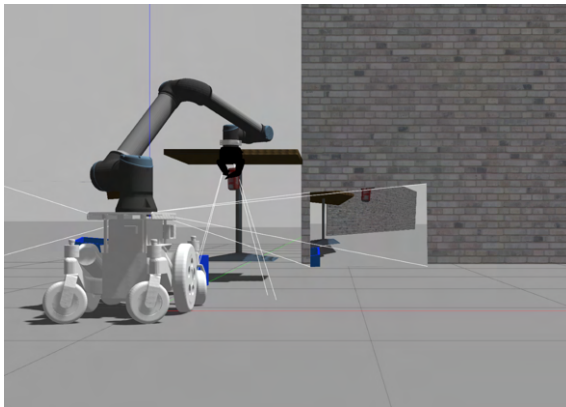


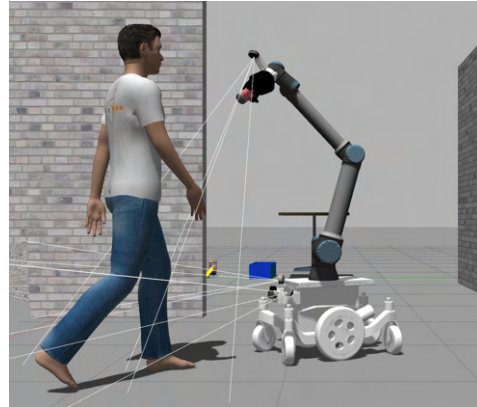
FIGURE 9.5: Amiga spawned in our pre-build map with amcl. The red arrows are particles that indicate its possible locations and will converge as Amiga moves and sensors update

Nonetheless, following the establishment of the maps, the probability approach AMCL can be used to locate the robot instead of odometry. The AMCL takes in a laser-based map, laser scans and transform messages, and outputs pose estimates. The particle filter is initialised according to the provided parameters, which can be managed again in a yaml file. Figure 9.5 illustrates the application of AMCL to our pre-built maps in the simulation environment. As already described in Background 9.1.1, the particle clouds are expected to shrink after Amiga moves and sensors update measurement through Bayes' theorem. For unknown initial positions, the AMCL node also provides a global localisation service wherein all particles are dispersed randomly through the map's free space and converge after locomotion.

Now that we have had all the inputs to the ROS navigation stack (Background 9.1.3), we can plan and execute safe paths on the map and move around Amiga in simulation. The global planner we used is navfn/NavfnROS [ros_navfn, 2014] and the local planner base_local_planner/TrajectoryPlanner ROS [ros_base_local_planner, 2014]. The path planning goal can be set in Rviz via a 2D navigation goal GUI or through a script's action client of the navigation stack. Figure 9.6 displays an example of a pick-and-handle task in a simulation environment – Amiga picks up a coke, travels to a predetermined destination on the map, detects a human, and hands him the coke.



(A) Amiga picks up a coke can



(B) Amiga handles the coke to a human

FIGURE 9.6: Amiga performing a pick-and-handle task in the simulation environment. Once a map is established, many locations can be defined on the map, and Amiga can plan and travel to them safely on demand

9.2.3 RTAB-Map and Multi-Sensor Inputs

Although the simulation environment is suitable for testing specific navigation components, its upper bound is determined by the computational power of a single laptop. Moreover, certain hardware-specific features cannot be tested, for example, the visual odometry functionality in ZED 2 camera's SDK. Therefore, another attempt was made with the real robot. Table 9.2 summaries our second strategy. A list of modifications have been made compared to our last trial, including odometry, sensing, mapping&localisation:

1. Odometry: changed from wheel odometry to ZED 2's visual odometry
2. Sensor: including both the camera and the LiDAR. The camera depth map is converted to laser scan and merged with the LiDAR laser scan
3. Mapping and localisation: adopted RTAB-Map instead of Gmapping and removed the use of amcl (as it is no longer supported)

Odometry Source	Sensor Source	Mapping	Localisation
ZED 2 Build-in	Camera&LiDAR	2D RTAB-Map	Visual Odometry

TABLE 9.2: Summary of the later strategy we adopted for the navigation module in real world

Figure 9.7 demonstrates the principle of RTAB-Map building on top of ROS navigation stack. The main difference between Gmapping and RTAB-Map is that RTAB-Map is essentially a Graph-SLAM approach, whereas GMapping relies on particle filter⁴. Moreover, RTAB-Map supports multiple sensor inputs (which need to be synchronised⁵) while Gmapping is limited to lasers.

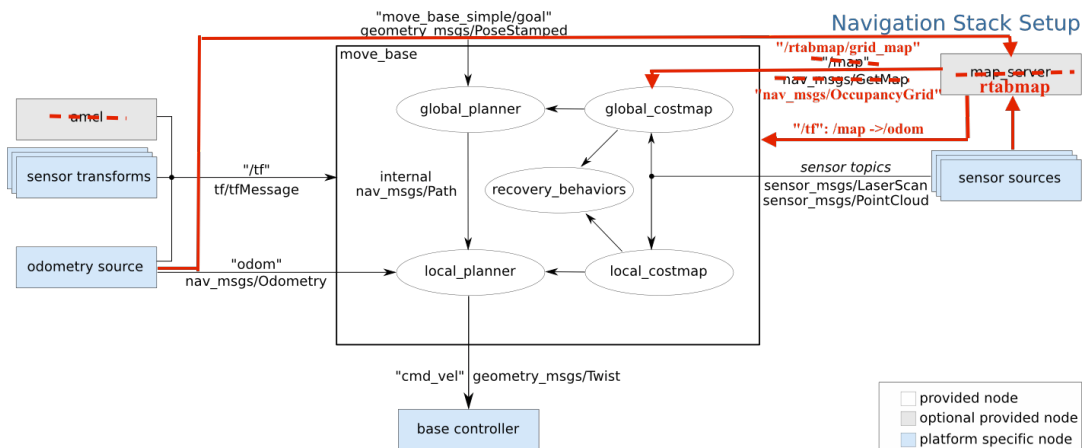


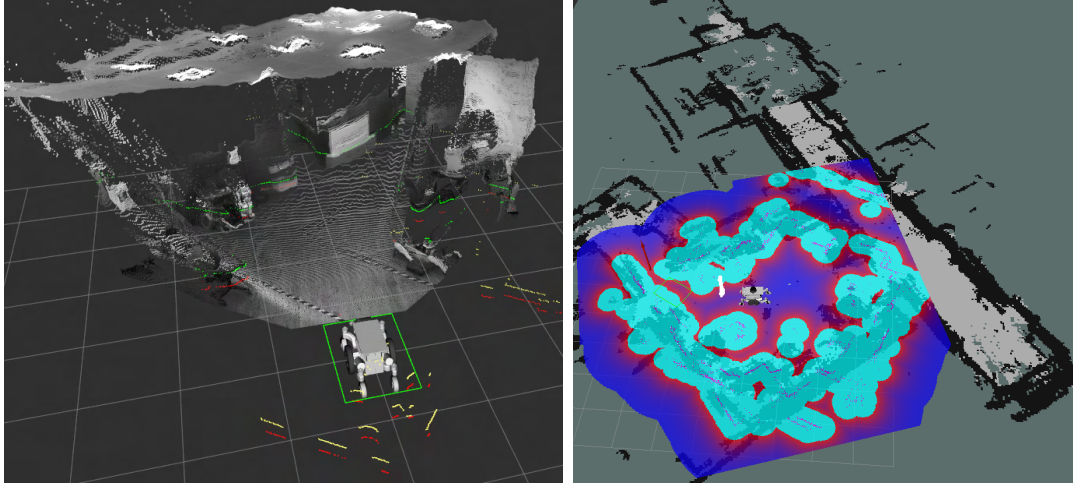
FIGURE 9.7: The mechanism of RTAB-Map, which builds on ROS navigation stack. The primary node replaces the map server and provides its own grid map along with a map to odom transform. The odom to base transform is published by the odometry source and used in the node for localisation

Figure 9.8 depicts the launching of the navigation stack in accordance with our new strategy. In 9.8a one can observe that the point cloud boundary aligns up with some of the laser scans in front of Amiga due to the conversion

⁴This explains why RTAB-Map does not support amcl as in Figure 9.7

⁵ROS provides two sorts of synchronisation: exact and approximate. Exact synchronisation of input topics requires precisely the same timestamp. By contrast, approximate synchronisation compares timestamps of the incoming topics and tries to synchronise all topics with a minimum delay error

from the depth images of the camera to scan messages. In addition, LiDAR sensors are used to generate scans from the back. A single scan was created after all scans were merged via the [ira_laser_tools, 2021] tools.



(A) Visualisation of Amiga's various sensor messages in real world. The messages include real world. The blue clusters represent the ZED 2's (on the wheelchair base) point cloud local cost map updated by the sensor messages and the merged laser scan

(B) Amiga spawned in a pre-built map in messages (Figure 9.8a) in real time

FIGURE 9.8: Amiga navigation launching in real world with multiple sensor sources (cameras and LiDARs)

In Figure 9.8b, a pre-built map⁶ from the lab was loaded to the navigation stack. As can be observed, the introduction of image sources caused some random noises to appear on the grid map (global cost map). Additionally, the blue cluster represents the local cost map, which is continually updated based on the sensor messages shown in Figure 9.8a. Similar to our previous case, we could employ a global&local planner plugin and relocate Amiga as dictated by the map.

Therefore, we can either predefine some positions on the map or target a 3D ROI coordinate generated by Amiga perception. The latter scenario requires that the coordinates in the camera frame be transformed into the map frame through the Odom frame. Tests indicate that ZED 2 camera odometry is being published frequently enough such that a "lookup would require extrapolation into the future" transformation error⁷ will not raise.

9.3 Architecture Summary and Comments

In summary, we have implemented a baseline navigation module as shown in 9.9. With this baseline module, Amiga can plan the paths to marked positions on any map or on-line locations determined by Amiga perception. Still,

⁶Built with RTAB-Map on the 10th floor of the EEE building by Goubard Cédric

⁷This happens if a non-static transform (such as map to odom) is not ready upon a listening request. Using external methods to compute visual odometry may lead to this error if the source node is not optimised sufficiently

a navigation explorer node needs to be implemented as Amiga perception remains local to the robot unless a ceiling-mounted global camera is employed (bird's-eye view).

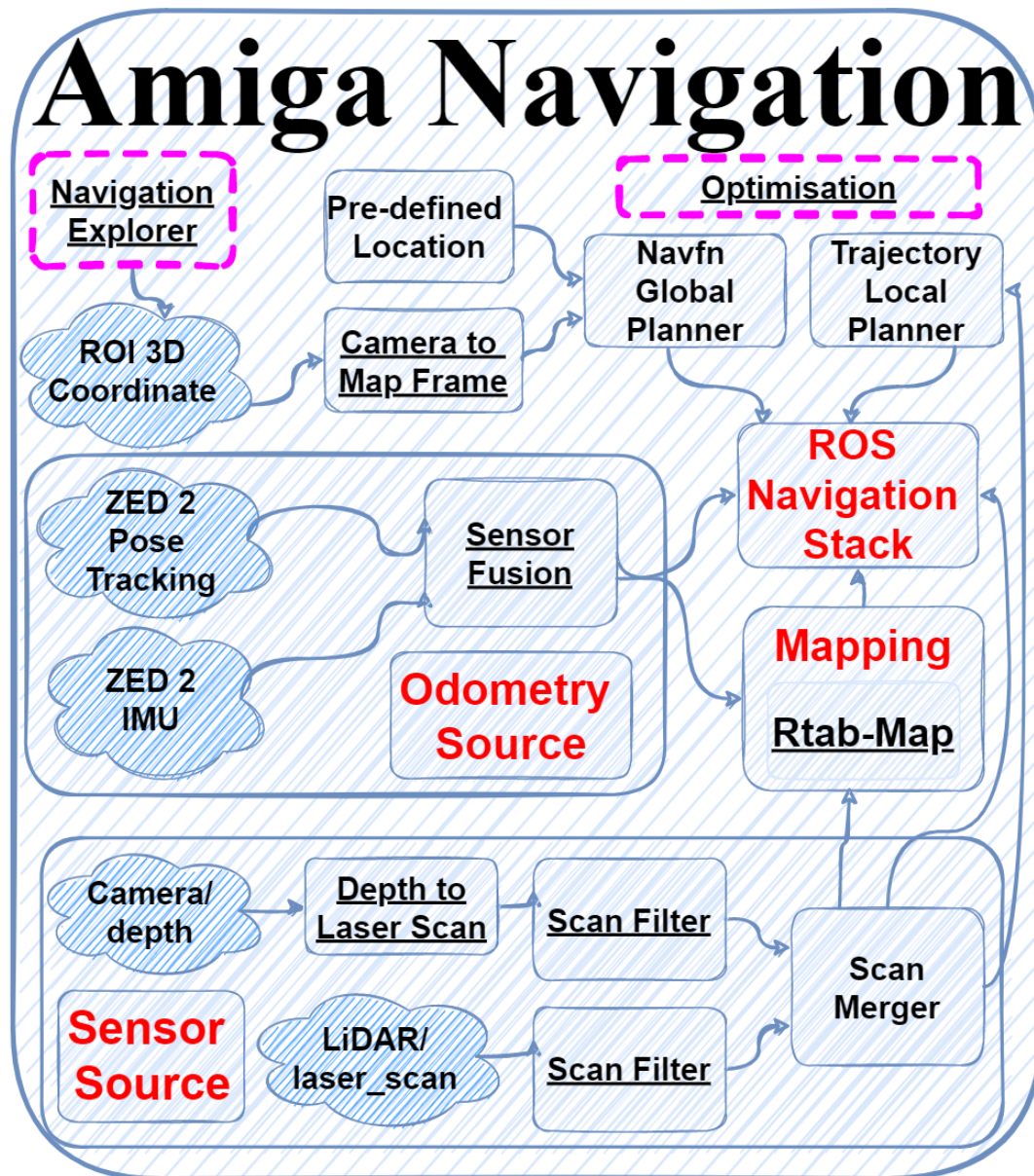


FIGURE 9.9: Amiga Navigation. Blue clouds represent ROS topics, while red markers highlight significant aspects. Future work is enclosed within pink dashed lines

Even though the navigation stack is a mature platform and highly compatible with Amiga, We must carry out plenty of source-adoption strategies and parameter optimisations. So far, the odometry source has been tested to be effective, but the sensor sources and the path planners are still subject to improvements in the future. Better maps (2D or 3D) need to be established via the chosen combinations of odometry and sensors, and various parameters (resolutions, footprints, tolerances, recoveries and so forth) within the

navigation stack will need to be optimised further in order to achieve precise path planning for Amiga. In light of our limited time, we will leave this task for a future date.

10 Testing and Evaluation

Considering the scope of the project, it is not feasible to evaluate each sub-module independently. In spite of this, due to active interactions among modules, most of them are automatically tested if they are dependent upon others. Manipulation, for example, is dependent on perception, grasping is dependent on manipulation, etc. Additionally, most core modules are already evaluated in the chapters in which they are found.

We present a brief evaluation of some interesting aspects of our system that have not been addressed before.

10.1 Plane detection

While the plane detection node we devised with python Open3D could not achieve real-time performance, the time performance with C++ PCL is tested to have at least 15 fps as found in table 10.1. The resolution mode suggests four available ones from ZED 2, and the input is the raw camera point cloud topics. The processing would be much quicker for object/region point clouds generated from cropped bounding boxes and aligned depth maps.

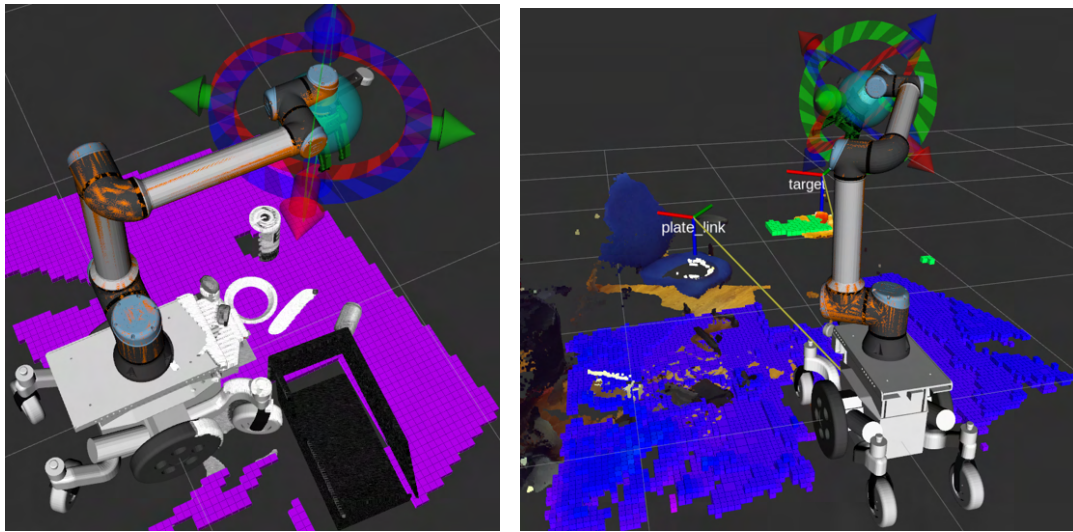
Resolution mode	FPS
HD2K	~ 15
HD1080	~ 30
HD720	~ 80
VGA	~ 100
SIM	~ 200

TABLE 10.1: Plane detection evaluation

10.2 Sim2real

While our simulation component can interact with the rest of the system for fast prototyping, there is essentially a sim2real gap. Figure 10.1 depicts such an example. In Figure 10.1a, the "all-planes-are-obstacle" features from our obstacle avoidance module perfectly add all ground as obstacles. On the other hand, as in Figure 10.1b we have some areas not covered as well as

some small errors due to sensor noises, which were not considered when we implemented the camera plugins in simulation.



(A) Plane detection sim

(B) Plane detection real

FIGURE 10.1: Evaluating plane detection

Grasping is maybe the most gaped manipulation task. Nonetheless, after optimising with the grasp plugin, we were able to simulate close-to-reality tasks as in Figure 10.2, where Amiga is trying to pour the coke into a bowl in simulation (although we did not implement liquid flowing).

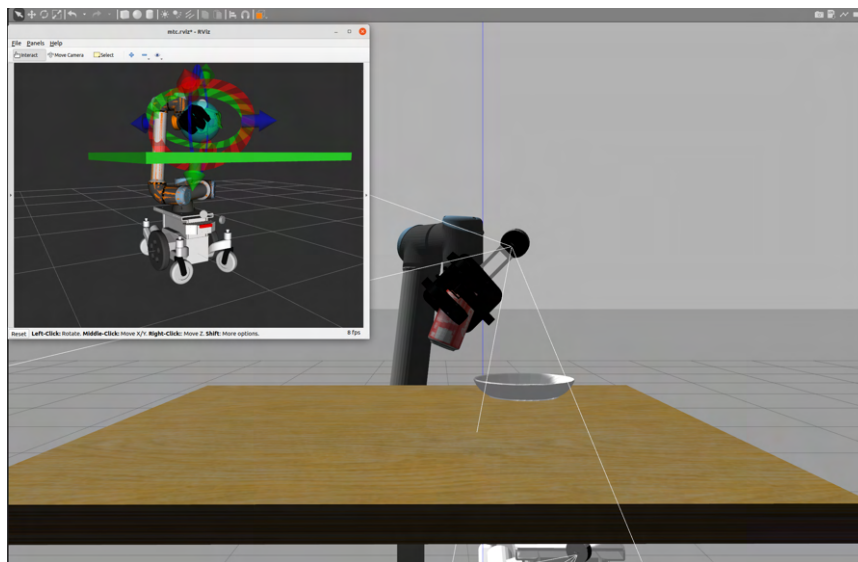


FIGURE 10.2: Amiga pouring in simulation

The presence of our simulation component is comforting because it can be used to test different IK solvers and path planners with little cost, whereas, with the real robot, it is usually expensive as one has to consider obstacles and constantly worry about unreasonable paths.

10.3 KitchenNet

In this section, we compare KitchenNet and YOLO V3 in detecting real-life kitchen objects. Because we only cared about real-life performances, we did not implement an evaluation/visualisation script for KitchenNet. The experiment was conducted by a human expert who decided qualitatively if an object could be reliably detected. A scene image is shown in 10.3 below.

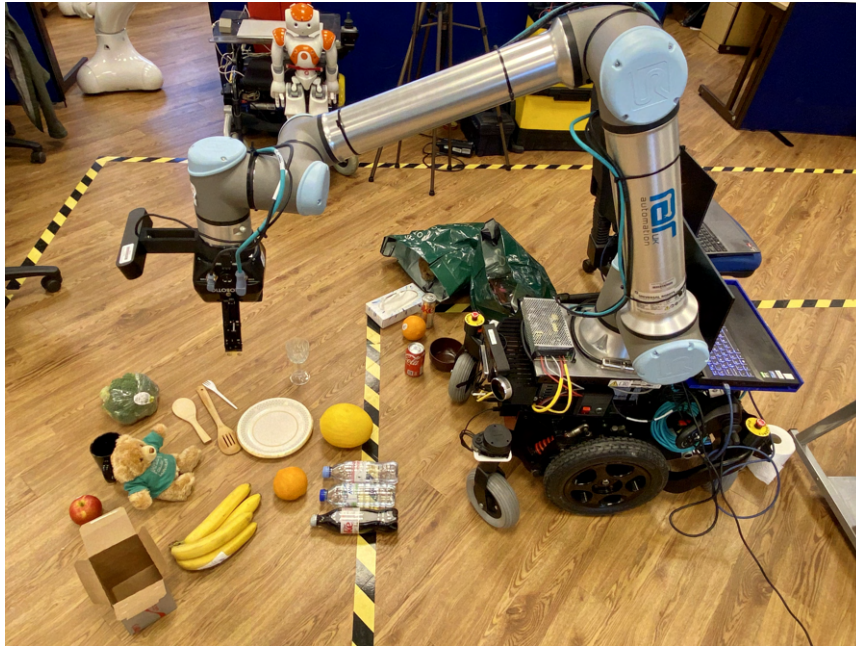


FIGURE 10.3: Evaluating Kitchen object detectors in real-life

The viewpoints were not adjusted, and the distance was kept at the position as in the image. The results were presented in table 10.2.

All object classes in the table are within the trained classes for KitchenNet. Due to a top-down view and a relatively high position, many small objects could not be detected well on this occasion. This was more so as KitchenNet was trained with SSD, which is not good at detecting small objects. Further, KitchenNet seems to have an issue detecting bottles for some reason, so we used YOLO v3 for such purposes. In addition, KitchenNet runs at 200 fps in Jetson Xavier, which was four times as fast as YOLO V3 on a 3070 GPU.

This experiment was inconclusive due to constraints on fixed height and viewpoint but should give us an initial sense.

10.4 Motion planning and Grasping

Testing of motion planning tasks is generally very expensive on Amiga. The reasons are summarised as follows:

1. "Arm controllers not running": we always encounter some unknown problems when launching the robot, suggesting the controllers were

Objects	KitchenNet	YOLO v3	Objects	KitchenNet	YOLO v3
Apple	✓	✓	Box	✓	✗
Banana	✓	✗	Lemon	✗	✗
Bottle	✗	✓	Plate	✓	✗
Cup	✗	✗	Coffee Cup	✗	✗
Fork	✗	✗	Chopsticks	✗	✗
Knife	✗	✗	Potato	✓	✗
Orange	✓	✓	Teddy bear	✓	✓
Broccoli	✗	✗	Towel	✓	✗
Bowl	✗	✗	Cutting board	✓	✗
Spoon	✓	✗	Egg	✗	✗
Tomato	✓	✗	Table	✗	✓

TABLE 10.2: Kitchen object evaluation

not running. While this could be resolved by relaunching multiple times, it was pretty time-consuming. Moreover, it was discovered that after hanging for 20 minutes, the arms controllers would again enter the "not running" phase stage. The tablet dashboard may have provided a service to disable this but may require some investigation.

2. "Safety modes": the arm would enter safety modes after a certain degree of violations of angles. Moreover, when the arm moves around, sometimes we are worried about the path it was executing and would push the emergency buttons. Recovering from these modes may take more than 30 minutes.
3. "Accidents": the robot experienced many accidents while executing a plan. We have experienced camera USB breaking, chargers breaking, and in some instances, it would enter a joint configuration which violates the initialisation of MoveIt's planning tree. Under this situation, we had to connect it to the tablet, enter the manual mode and recover it back to a feasible joint configuration state.

We evaluate the motion planning module's ability to perform high DoF grasping as in table 10.3. Most of the errors are kinematical causes. In addition, because many of the experiments were conducted on the ground and added as obstacles, we also encountered issues in which a full plan was not executed, resulting in the gripper hanging in mid-air.

As a matter of fact, most of the failures in high DoF grasping were either caused by collision or human intervention—the essential issues down to the

unreasonable paths generated by our current IK solvers and path planners. Therefore, the results presented in the table would remain inconclusive for higher DoF grasps. On the other hand, as the degree of freedom decreases, its reliance on IK solvers and path planners also decreases, resulting in more successful grasps. More testing would be needed to unravel the potential of the grasping module.

Type	Objects	Success over trials	Reasonable paths
3 DoF grasp	apple and more	8/8	6/8
3 DoF grasp	bottle	4/4	4/4
4 DoF grasp	coffee cup	4/4	4/4
4 DoF grasp	teaspoon	2/4	4/4
5 DoF grasp	knife	1/4	2/4
5 DoF grasp	bowl	3/6	3/6
6 DoF grasp	kitchen roll	1/2	2/2
6 DoF grasp	cloth	1/4	1/4
6 DoF grasp	bowl	3/10	3/10
6 DoF grasp	spatula	2/4	1/4

TABLE 10.3: Grasp evaluation

11 Conclusions and Further Work

In conclusion, we have proposed a modular architecture for Amiga to carry out kitchen activities. The system covers simulation, perception, manipulation, grasping and navigation and satisfies the following properties:

- *Generality* – The system possesses the ability to perform various tasks, gain new skills, apply these skills in novel situations, adapt continuously and improve performance.
- *Exensibility* – The system is designed with high cohesion and low coupling such that adding more functions within the same framework is straightforward and effortless.
- *Complexity* – The system orchestrates algorithmic capabilities for perception, manipulation, navigation, controlling, and planning by using multiple hardware components for sensing, manipulating, and locomotion.
- *Distributability* – The system is portable and can be tested to achieve optimal utilization of computation power across different types of computers used in the robot (laptops, embedded devices, etcetera) in different environments.

However, the more generally the framework, the more uncertainty it entails. More straightforward methods have more trust, whereas complicated methods have low trust, and good results can be hard to reproduce. For example, MoveIt is a compelling framework, but many features remain basic, such as primitive interactions, adding primitives to collision scenes, and grasping a primitive object. While we expanded these features by introducing deep CNN algorithms, the system's stability is no longer guaranteed. Effectively, the gap of trust between a human and a robot is enlarged.

11.1 Limitations

Many are already discussed in the relevant chapter (simulation, perception, motion planning and grasping). Generally speaking, the main limitations are:

1. Object detection models are trained on general pictures rather than first-person view. Image segmentation may be better candidate than object detection (but harder to produce)

2. Deep grasping methods cannot guarantee the object will be grasped at a desired pose to handle subsequent tasks
3. Motion planning modules have not been optimised in terms of planners, IK solvers and collision checkers
4. Navigation modules have not been optimised in terms of mapping and planners

11.2 Future Work

The majority of these have already been addressed in the relevant implementation sections. Numerous work can be done to improve our existing framework; some examples are:

1. Re-print a 3D clamp as it leads to occlusion right now
2. Switch to better IKs such as IK Fast and relaxed_IK and more advanced planners
3. Exhaustively testing the high DoF deep grasping methods
4. Study Grasp-motion affiliation to perform more complex tasks
5. Define better trajectories and record video Datasets such as to establish perception algorithms benchmarks
6. Fuse multi-viewpoint cloud and design a trajectory to complete the shape of the partial point cloud
7. Human robot interaction such as speech interactions
8. Incorporate learning methods such as imitation learning and reinforcement learning
9. Improve Amiga Simulation by finishing the left-overs as in [5.3](#)
10. Improve Amiga Perception by finishing the left-overs as in [6.3](#)
11. Improve Amiga Manipulation by finishing the left-overs as in [7.3](#)
12. Improve Amiga Grasping by finishing the left-overs as in [8.3](#)
13. Improve Amiga Navigation by finishing the left-overs as in [9.3](#)

12 User Guide

Some partial code available at https://github.com/yw14218/amiga_extra
 Note due to multiple components involved in this project, the code base is under active cleaning and a README will be added in the near future. Note that some dependencies and modified package were located on Amiga's laptop. Please consult Personal Robotics Lab, Imperial College London for permissions to access the full packages as well as the dockerfiles.

To launch Amiga:

```
# Noetic
```

```
## Using the onboard Jetsons
```

The best way to control the robot is to run:

- the realsense driver on the base Jetson
(can also run SLAM and visual odom if needed)
- the arm driver, ``MoveIt``, SLAM, visual odom, move_base and ``rviz`` on the laptop
- the object detection module on the arm Jetson

To do so, use the Docker images.

The ROS core should be on the base Jetson.

```
### Installation of the navigation stack
```

```
ssh JETSON_USER@JETSON_IP # SSH to the Jetson
```

```
git clone git@github.com:ImperialCollegeLondon/rnet-wheelchair-docker.
```

```
git --recurse-submodules
```

```
cd rnet-wheelchair-docker
```

```
# You don't need to set up your .env file here,
```

```
since one will be copied from the external laptop at runtime
```

```
# Login dockerhub (you will need
```

```
some prebuilt images from the lab)
```

```
docker login
```

```
### Installation on the Laptop
```

```
# Clone the arm driver
```

```
git clone -b noetic git@github.com:
```

```
NicoLingg/ur10e_robotiq.git
--recurse-submodules
cd ur10e_robotiq
vim .env # Set your IP here, as well as the Jetson's,
and select what you want to run

# Login dockerhub (you will need some prebuilt images from the lab)
docker login

+

### Installation on the arm Jetson for object detection

ssh JETSON_USER@JETSON_IP # SSH to the Jetson (the other one)
# First, we clone the repo
git clone -b noetic git@github.com:NicoLingg/
ur10e_robotiq.git --recurse-submodules
cd ur10e_robotiq

# You don't need to set up your .env file here, since one
will be copied from the external laptop at runtime

# Login dockerhub (you will need some prebuilt images from the lab)
docker login

**NB**: you will need to run the node once, to optimise
the model for the Jetson's GPU. This will generate a set
of files (normally in ~/usr/local/bin/networks/SSD-Mobilenet-v2/).
Copy the whole folder outside of the container, and mount
it at runtime to avoid calibrating each time you run the process:

cp -r [FOLDER_YOU_RETRIEVED_FROM_THE_CONTAINER] .calibrated_models/

# This folder should have been automatically
created when you built the image

### Running everything

Everything can be launched from the laptop
(from the root of this repo, ur10e_robotiq).

Run in a first terminal:
make restart-navigation
```

```
# this is run over SSH, so CTRL-C won't really stop it.  
Use make stop-navigation instead
```

This should reset the raspberry pi, and start the mapping.

Then, in another terminal:

```
make restart-obj-detect  
# this is run over SSH, so CTRL-C won't really stop it.  
Use make stop-obj-detect instead
```

Finally, in a third terminal:

```
make run-external
```

Desktop Install (mainly for simulation)

Prerequisites:

- ROS Noetic installed
- A ROS workspace initiated
(I'm using [catkin tools](<https://catkin-tools.readthedocs.io/en/latest/installing.html>),
if you're not just replace `catkin
build` with `catkin_make`)

Installation

```
cd [YOUR_ROS_WORKSPACE]/src  
git clone -b noetic git@github.com:NicoLingg/ur10e_robotiq.git  
rosdep install --roscdistro noetic --ignore-src --from-paths .  
catkin build amiga_sim  
source ../devel/setup.bash # Or .zsh
```

Running the simulation

```
# This launches the simulation.  
Use the MoveIt interface in RViz to move the arm.  
roslaunch amiga_sim gazebo_basic.launch
```

```
# If you want to move the base, you can use  
roslaunch teleop_twist_keyboard teleop_twist_keyboard.py
```

```
cmd_vel:=/arnie_base_controller/cmd_vel
```

A Appendix

A.1 Amiga Simulation

A.1.1 Teleop and Gazebo Worlds

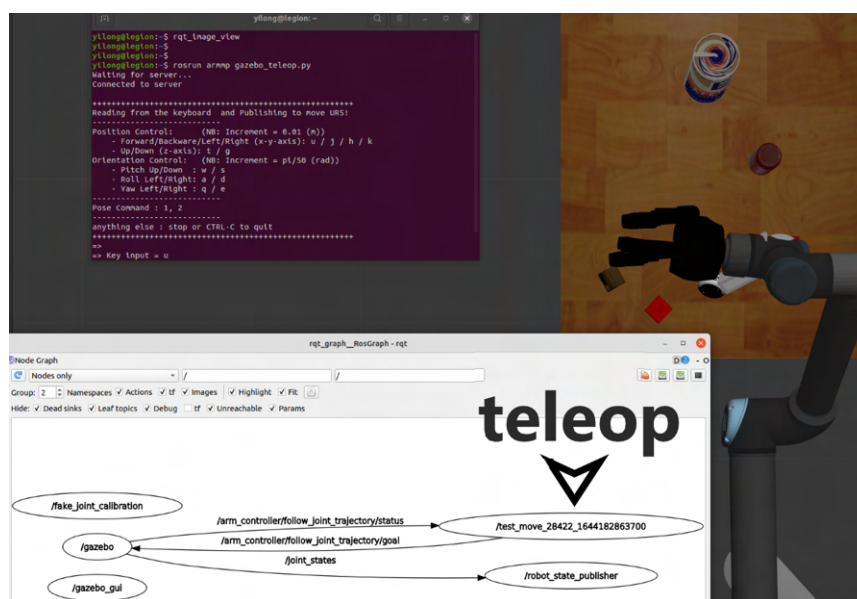
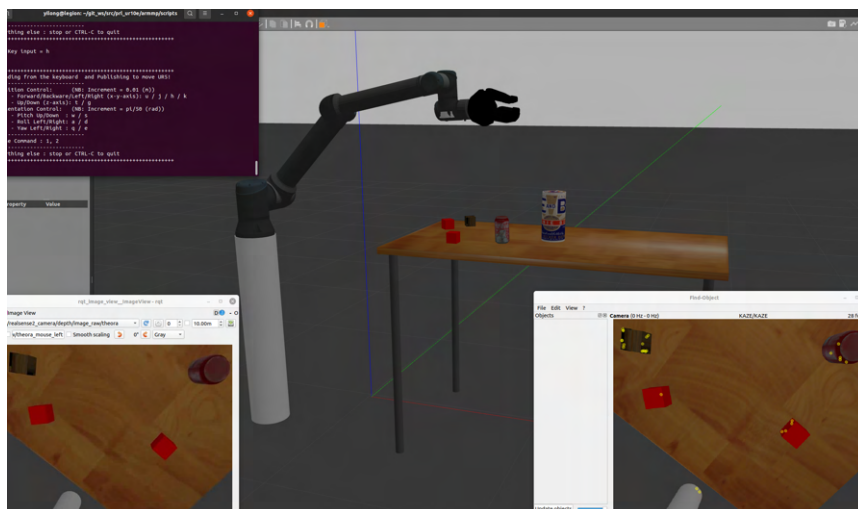
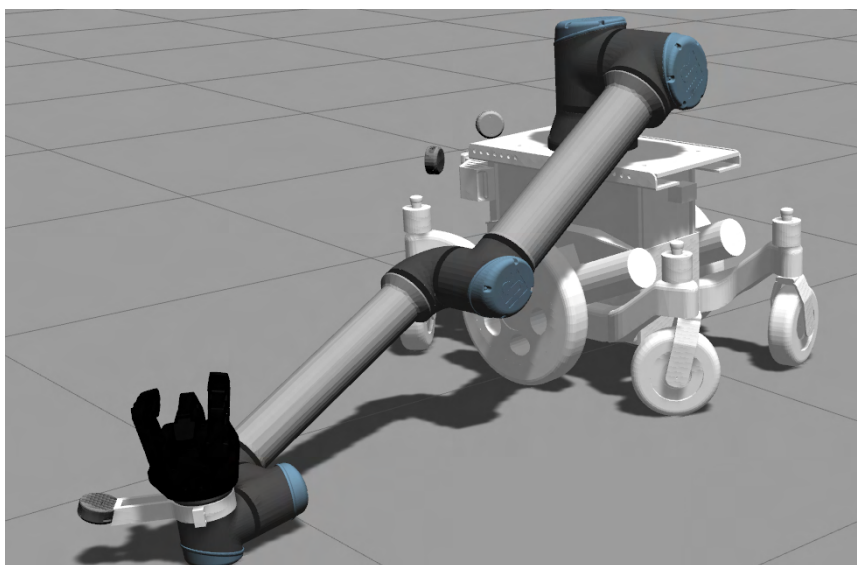


FIGURE A.1: Teleops for controlling UR10e arms in Gazebo



(A) Early implementation with only the arm and gripper



(B) Furbished rendering and included wheelchair base. Many bugs are to be fixed (gripper shaking, arm crumbling and etc)



(C) Final platform with rich kitchen assistant elements

FIGURE A.2: Amiga simulation at various stages of the project

A.1.2 Intrinsic and Hand-Eye calibration in Gazebo

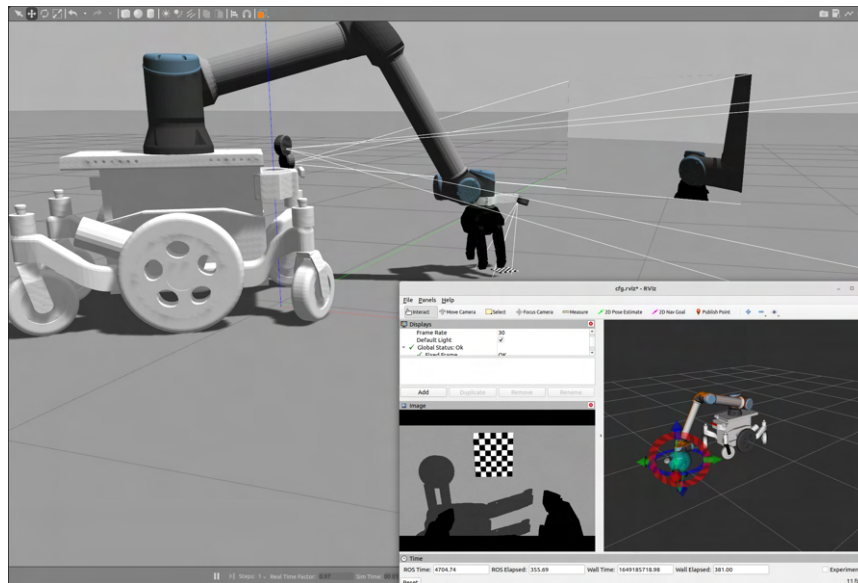


FIGURE A.3: Camera intrinsic calibration in Gazebo

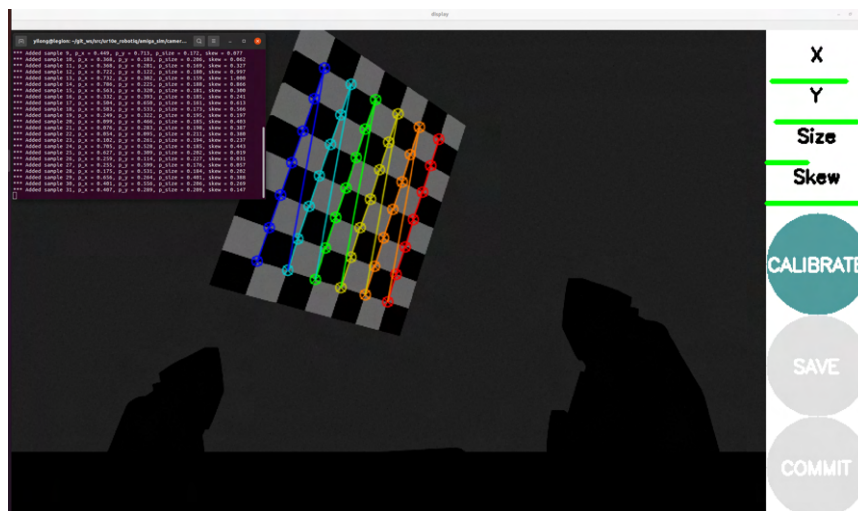


FIGURE A.4: Camera intrinsic calibration in Gazebo

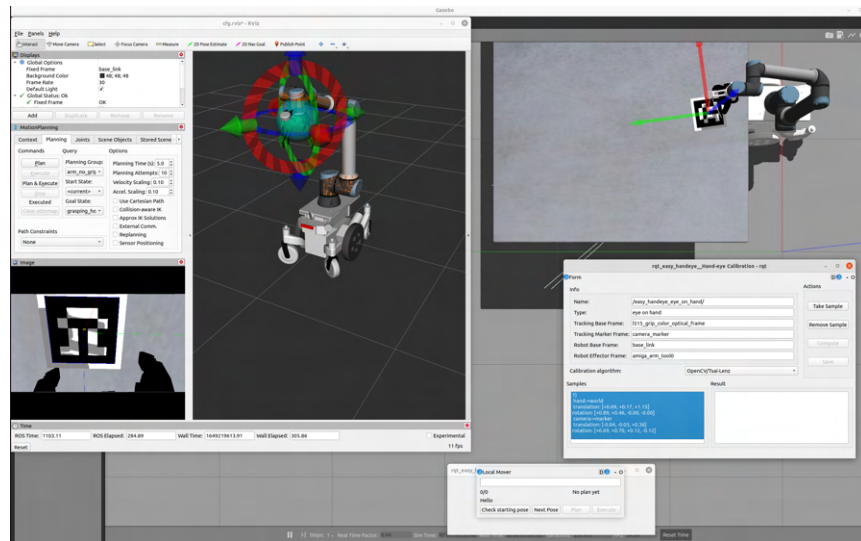


FIGURE A.5: Hand-Eye calibration in Gazebo

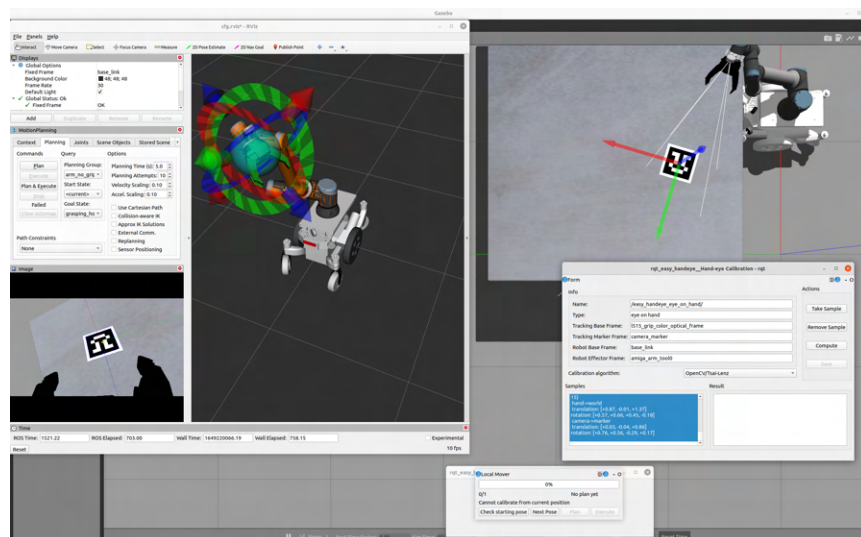


FIGURE A.6: Hand-Eye calibration in Gazebo

A.1.3 Arm and Gripper Controllers

```

1 controller_list:
2   - name: arm_controller
3     action_ns: follow_joint_trajectory
4     type: FollowJointTrajectory
5     default: true
6     joints:
7       - shoulder_pan_joint
8       - shoulder_lift_joint
9       - elbow_joint
10      - wrist_1_joint
11      - wrist_2_joint
12      - wrist_3_joint
13   - name: gripper
14     action_ns: follow_joint_trajectory
15     type: FollowJointTrajectory
16     default: true
17     joints:
18       - gripper_palm_finger_1_joint
19       - gripper_finger_1_joint_1
20       - gripper_finger_1_joint_2
21       - gripper_finger_1_joint_3
22       - gripper_palm_finger_2_joint
23       - gripper_finger_2_joint_1
24       - gripper_finger_2_joint_2
25       - gripper_finger_2_joint_3
26       - gripper_finger_middle_joint_1
27       - gripper_finger_middle_joint_2
28       - gripper_finger_middle_joint_3
29
30 arm_controller:
31   type: position_controllers/JointTrajectoryController
32   joints:
33     - shoulder_pan_joint
34     - shoulder_lift_joint
35     - elbow_joint
36     - wrist_1_joint
37     - wrist_2_joint
38     - wrist_3_joint
39   constraints:
40     goal_time: 0.6
41     stopped_velocity_tolerance: 0.05
42     shoulder_pan_joint: {trajectory: 0.1, goal: 0.1}
43     shoulder_lift_joint: {trajectory: 0.1, goal: 0.1}
44     elbow_joint: {trajectory: 0.1, goal: 0.1}
45     wrist_1_joint: {trajectory: 0.1, goal: 0.1}
46     wrist_2_joint: {trajectory: 0.1, goal: 0.1}
47     wrist_3_joint: {trajectory: 0.1, goal: 0.1}
48   stop_trajectory_duration: 0.5
49   state_publish_rate: 25
50   action_monitor_rate: 10
51   joint_group_position_controller:
52     type: position_controllers/JointGroupPositionController
53     joints:
54       - shoulder_pan_joint
55       - shoulder_lift_joint
56       - elbow_joint
57       - wrist_1_joint
58       - wrist_2_joint
59       - wrist_3_joint
60
61 gripper:
62   type: effort_controllers/JointTrajectoryController
63   joints:
64     - gripper_finger_1_joint_1
65     - gripper_finger_1_joint_2
66     - gripper_finger_1_joint_3
67     - gripper_finger_2_joint_1
68     - gripper_finger_2_joint_2
69     - gripper_finger_2_joint_3
70     - gripper_finger_middle_joint_1
71     - gripper_finger_middle_joint_2
72     - gripper_finger_middle_joint_3
73     - gripper_palm_finger_1_joint
74     - gripper_palm_finger_2_joint
75   constraints:
76     goal_time: 0.6
77     stopped_velocity_tolerance: 0.05
78     gripper_finger_1_joint_1: {trajectory: 0.005, goal: 0.005}
79     gripper_finger_1_joint_2: {trajectory: 0.005, goal: 0.005}
80     gripper_finger_1_joint_3: {trajectory: 0.005, goal: 0.005}
81     gripper_finger_2_joint_1: {trajectory: 0.005, goal: 0.005}
82     gripper_finger_2_joint_2: {trajectory: 0.005, goal: 0.005}
83     gripper_finger_2_joint_3: {trajectory: 0.005, goal: 0.005}
84     gripper_finger_middle_joint_1: {trajectory: 0.005, goal: 0.005}
85     gripper_finger_middle_joint_2: {trajectory: 0.005, goal: 0.005}
86     gripper_finger_middle_joint_3: {trajectory: 0.005, goal: 0.005}
87     gripper_palm_finger_1_joint: {trajectory: 0.005, goal: 0.005}
88     gripper_palm_finger_2_joint: {trajectory: 0.005, goal: 0.005}
89   stop_trajectory_duration: 0.5
90   state_publish_rate: 25
91   action_monitor_rate: 10
92   gains: # Required because we're controlling an effort interface
93     gripper_finger_1_joint_1: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
94     gripper_finger_1_joint_2: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
95     gripper_finger_1_joint_3: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
96     gripper_finger_2_joint_1: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
97     gripper_finger_2_joint_2: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
98     gripper_finger_2_joint_3: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
99     gripper_finger_middle_joint_1: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
100    gripper_finger_middle_joint_2: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
101    gripper_finger_middle_joint_3: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
102    gripper_palm_finger_1_joint: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}
103    gripper_palm_finger_2_joint: {p: 1, d: 0.5, i: 0, f: 0, clamp: 0}

```

FIGURE A.7: Configuration files developed for Amiga's integration of MoveIt to Gazebo

A.2 Amiga Perception

A.2.1 Camera Topics

For L515:

- RGB Image: /l515_grip/color/image_raw
- Depth Image: /l515_grip/aligned_depth_to_color/image_raw
- Point Cloud: /l515_grip/depth/color/points
- Camera Parameters: /l515_grip/color/camera_info

For ZED2:

- RGB Image: /zed2_grip/rgb/image_rect_color
- Depth Image: /zed2_grip/depth/depth_registered
- Point Cloud: /zed2_grip/point_cloud/cloud_registered
- Camera Parameters: /zed2_grip/rgb/camera_info

A.2.2 3D Coordinate Mapping

Pointcloud Approach: this method uses the pointcloud messages published by the cameras to directly compute 3D coordinate xyz

```

import struct

def uv2xyz(u, v, pc_msg):
    datatype = {1:1, 2:1, 3:2, 4:2, 5:4, 6:4, 7:4, 8:8}
    arrayPosition = v*pc_msg.row_step + u*pc_msg.point_step
    # X has an offset of 0
    pos_x = arrayPosition + pc_msg.fields[0].offset
    len_x = datatype[pc_msg.fields[0].datatype]
    # Y has an offset of 4
    pos_y = arrayPosition + pc_msg.fields[1].offset
    len_y = datatype[pc_msg.fields[1].datatype]
    # Z has an offset of 8
    pos_z = arrayPosition + pc_msg.fields[2].offset
    len_z = datatype[pc_msg.fields[2].datatype]

    try:
        # read 4 bytes as a float number
        x = struct.unpack('f', pc_msg.data[pos_x: pos_x+len_x])[0]
        y = struct.unpack('f', pc_msg.data[pos_y: pos_y+len_y])[0]
        z = struct.unpack('f', pc_msg.data[pos_z: pos_z+len_z])[0]
        return [x,y,z]
    except:
        return None

```

A.2.3 Object Detection

COCO Kitchen-relevant Classes:

```

id_dict = {
    'bottle' : 44,
    'plate' : 45,
    'wine glass' : 46,
    'cup' : 47,
    'fork' : 48,
    'knife' : 49,
    'spoon' : 50,
    'bowl' : 51,
    'banana' : 52,
    'apple' : 53,
    'sandwich' : 54,
    'orange' : 55,
    'broccoli' : 56,
    'carrot' : 57,
    'hot dog' : 58,
    'pizza' : 59,
    'donut' : 60,
    'cake' : 61,

```

```
'blender' : 83,  
}
```

KitchenNet Dataset Training Distribution:

```
-----  
'train' set statistics  
-----
```

```
Image count: 71425  
Bounding box count: 128656  
Bounding box distribution:  
Chair: 8000/128656 = 0.06  
Beer: 8000/128656 = 0.06  
Drink: 8000/128656 = 0.06  
Vegetable: 8000/128656 = 0.06  
Wine glass: 8000/128656 = 0.06  
Bottle: 8000/128656 = 0.06  
Human hand: 8000/128656 = 0.06  
Person: 8000/128656 = 0.06  
Table: 8000/128656 = 0.06  
Orange: 6186/128656 = 0.05  
Tomato: 6071/128656 = 0.05  
Box: 5300/128656 = 0.04  
Plate: 5221/128656 = 0.04  
Coffee cup: 5068/128656 = 0.04  
Bowl: 4482/128656 = 0.03  
Apple: 3622/128656 = 0.03  
Tin can: 2749/128656 = 0.02  
Egg: 1752/128656 = 0.01  
Lemon: 1698/128656 = 0.01  
Spoon: 1644/128656 = 0.01  
Fork: 1609/128656 = 0.01  
Banana: 1574/128656 = 0.01  
Carrot: 1440/128656 = 0.01  
Teddy bear: 1425/128656 = 0.01  
Tea: 1301/128656 = 0.01  
Broccoli: 1115/128656 = 0.01  
Watermelon: 753/128656 = 0.01  
Chopsticks: 612/128656 = 0.00  
Potato: 596/128656 = 0.00  
Kitchen utensil: 546/128656 = 0.00  
Frying pan: 376/128656 = 0.00  
Kitchen knife: 345/128656 = 0.00  
Towel: 308/128656 = 0.00  
Drinking straw: 291/128656 = 0.00  
Cutting board: 213/128656 = 0.00  
Milk: 193/128656 = 0.00  
Beaker: 166/128656 = 0.00
```

```

2022-05-22 16:10:27 - Epoch: 99, Step: 1510/2019, Avg Loss: 3.9256, Avg Regression Loss 1.0712, Avg Classification Loss: 2.8544
2022-05-22 16:10:34 - Epoch: 99, Step: 1520/2019, Avg Loss: 4.1254, Avg Regression Loss 1.1163, Avg Classification Loss: 2.8091
2022-05-22 16:10:42 - Epoch: 99, Step: 1530/2019, Avg Loss: 4.0958, Avg Regression Loss 1.1505, Avg Classification Loss: 2.7453
2022-05-22 16:10:50 - Epoch: 99, Step: 1540/2019, Avg Loss: 3.9531, Avg Regression Loss 1.2270, Avg Classification Loss: 2.7261
2022-05-22 16:10:58 - Epoch: 99, Step: 1550/2019, Avg Loss: 3.6718, Avg Regression Loss 1.0717, Avg Classification Loss: 2.6001
2022-05-22 16:11:05 - Epoch: 99, Step: 1560/2019, Avg Loss: 3.6316, Avg Regression Loss 1.0623, Avg Classification Loss: 2.5693
2022-05-22 16:11:13 - Epoch: 99, Step: 1570/2019, Avg Loss: 4.1128, Avg Regression Loss 1.2571, Avg Classification Loss: 2.8557
2022-05-22 16:11:20 - Epoch: 99, Step: 1580/2019, Avg Loss: 3.9541, Avg Regression Loss 1.2232, Avg Classification Loss: 2.7217
2022-05-22 16:11:28 - Epoch: 99, Step: 1590/2019, Avg Loss: 3.7924, Avg Regression Loss 1.1368, Avg Classification Loss: 2.6557
2022-05-22 16:11:36 - Epoch: 99, Step: 1600/2019, Avg Loss: 4.1445, Avg Regression Loss 1.2938, Avg Classification Loss: 2.8507
2022-05-22 16:11:44 - Epoch: 99, Step: 1610/2019, Avg Loss: 3.9411, Avg Regression Loss 1.1609, Avg Classification Loss: 2.7801
2022-05-22 16:11:51 - Epoch: 99, Step: 1620/2019, Avg Loss: 4.1790, Avg Regression Loss 1.2364, Avg Classification Loss: 2.9427
2022-05-22 16:11:59 - Epoch: 99, Step: 1630/2019, Avg Loss: 4.0305, Avg Regression Loss 1.2090, Avg Classification Loss: 2.7706
2022-05-22 16:12:07 - Epoch: 99, Step: 1640/2019, Avg Loss: 4.0831, Avg Regression Loss 1.2160, Avg Classification Loss: 2.7071
2022-05-22 16:12:15 - Epoch: 99, Step: 1650/2019, Avg Loss: 3.8599, Avg Regression Loss 1.1858, Avg Classification Loss: 2.6742
2022-05-22 16:12:23 - Epoch: 99, Step: 1660/2019, Avg Loss: 3.8857, Avg Regression Loss 1.1407, Avg Classification Loss: 2.7451
2022-05-22 16:12:31 - Epoch: 99, Step: 1670/2019, Avg Loss: 3.8955, Avg Regression Loss 1.1910, Avg Classification Loss: 2.7044
2022-05-22 16:12:38 - Epoch: 99, Step: 1680/2019, Avg Loss: 3.8693, Avg Regression Loss 1.1417, Avg Classification Loss: 2.7275
2022-05-22 16:12:46 - Epoch: 99, Step: 1690/2019, Avg Loss: 4.0490, Avg Regression Loss 1.3048, Avg Classification Loss: 2.7442
2022-05-22 16:12:54 - Epoch: 99, Step: 1700/2019, Avg Loss: 3.9510, Avg Regression Loss 1.3011, Avg Classification Loss: 2.6499
2022-05-22 16:13:01 - Epoch: 99, Step: 1710/2019, Avg Loss: 4.0445, Avg Regression Loss 1.1479, Avg Classification Loss: 2.8966
2022-05-22 16:13:09 - Epoch: 99, Step: 1720/2019, Avg Loss: 4.2335, Avg Regression Loss 1.4220, Avg Classification Loss: 2.8115
2022-05-22 16:13:16 - Epoch: 99, Step: 1730/2019, Avg Loss: 3.8270, Avg Regression Loss 1.1467, Avg Classification Loss: 2.6802
2022-05-22 16:13:23 - Epoch: 99, Step: 1740/2019, Avg Loss: 3.9027, Avg Regression Loss 1.2116, Avg Classification Loss: 2.6911
2022-05-22 16:13:31 - Epoch: 99, Step: 1750/2019, Avg Loss: 4.0757, Avg Regression Loss 1.2688, Avg Classification Loss: 2.8069
2022-05-22 16:13:38 - Epoch: 99, Step: 1760/2019, Avg Loss: 3.9063, Avg Regression Loss 1.2466, Avg Classification Loss: 2.6597
2022-05-22 16:13:47 - Epoch: 99, Step: 1770/2019, Avg Loss: 3.7820, Avg Regression Loss 1.2140, Avg Classification Loss: 2.5680
2022-05-22 16:13:54 - Epoch: 99, Step: 1780/2019, Avg Loss: 4.0549, Avg Regression Loss 1.2557, Avg Classification Loss: 2.7993
2022-05-22 16:14:02 - Epoch: 99, Step: 1790/2019, Avg Loss: 3.7684, Avg Regression Loss 1.1338, Avg Classification Loss: 2.6347
2022-05-22 16:14:10 - Epoch: 99, Step: 1800/2019, Avg Loss: 3.8579, Avg Regression Loss 1.1617, Avg Classification Loss: 2.6962
2022-05-22 16:14:17 - Epoch: 99, Step: 1810/2019, Avg Loss: 3.9156, Avg Regression Loss 1.2413, Avg Classification Loss: 2.6743
2022-05-22 16:14:25 - Epoch: 99, Step: 1820/2019, Avg Loss: 3.8528, Avg Regression Loss 1.1116, Avg Classification Loss: 2.7412
2022-05-22 16:14:32 - Epoch: 99, Step: 1830/2019, Avg Loss: 3.8191, Avg Regression Loss 1.1628, Avg Classification Loss: 2.6563
2022-05-22 16:14:39 - Epoch: 99, Step: 1840/2019, Avg Loss: 3.7914, Avg Regression Loss 1.1598, Avg Classification Loss: 2.6316
2022-05-22 16:14:46 - Epoch: 99, Step: 1850/2019, Avg Loss: 3.7857, Avg Regression Loss 1.0656, Avg Classification Loss: 2.7202
2022-05-22 16:14:56 - Epoch: 99, Step: 1860/2019, Avg Loss: 4.0830, Avg Regression Loss 1.2737, Avg Classification Loss: 2.7293
2022-05-22 16:15:03 - Epoch: 99, Step: 1870/2019, Avg Loss: 3.9683, Avg Regression Loss 1.2531, Avg Classification Loss: 2.7152
2022-05-22 16:15:13 - Epoch: 99, Step: 1880/2019, Avg Loss: 3.9111, Avg Regression Loss 1.1041, Avg Classification Loss: 2.8070
2022-05-22 16:15:21 - Epoch: 99, Step: 1890/2019, Avg Loss: 4.0460, Avg Regression Loss 1.2293, Avg Classification Loss: 2.8167
2022-05-22 16:15:28 - Epoch: 99, Step: 1900/2019, Avg Loss: 3.7971, Avg Regression Loss 1.1723, Avg Classification Loss: 2.6248
2022-05-22 16:15:37 - Epoch: 99, Step: 1910/2019, Avg Loss: 3.8868, Avg Regression Loss 1.1394, Avg Classification Loss: 2.7474
2022-05-22 16:15:45 - Epoch: 99, Step: 1920/2019, Avg Loss: 3.8759, Avg Regression Loss 1.2046, Avg Classification Loss: 2.6713
2022-05-22 16:15:58 - Epoch: 99, Step: 1930/2019, Avg Loss: 4.0962, Avg Regression Loss 1.2090, Avg Classification Loss: 2.8872
2022-05-22 16:16:06 - Epoch: 99, Step: 1940/2019, Avg Loss: 3.8773, Avg Regression Loss 1.2242, Avg Classification Loss: 2.6532
2022-05-22 16:16:14 - Epoch: 99, Step: 1950/2019, Avg Loss: 4.0908, Avg Regression Loss 1.2931, Avg Classification Loss: 2.7977
2022-05-22 16:16:23 - Epoch: 99, Step: 1960/2019, Avg Loss: 3.6721, Avg Regression Loss 1.0584, Avg Classification Loss: 2.6139
2022-05-22 16:16:32 - Epoch: 99, Step: 1970/2019, Avg Loss: 3.8211, Avg Regression Loss 1.1633, Avg Classification Loss: 2.6578
2022-05-22 16:16:40 - Epoch: 99, Step: 1980/2019, Avg Loss: 3.8815, Avg Regression Loss 1.1010, Avg Classification Loss: 2.7805
2022-05-22 16:16:47 - Epoch: 99, Step: 1990/2019, Avg Loss: 4.0214, Avg Regression Loss 1.2512, Avg Classification Loss: 2.7701
2022-05-22 16:16:55 - Epoch: 99, Step: 2000/2019, Avg Loss: 4.1309, Avg Regression Loss 1.3724, Avg Classification Loss: 2.7584
2022-05-22 16:17:04 - Epoch: 99, Step: 2010/2019, Avg Loss: 3.9195, Avg Regression Loss 1.1921, Avg Classification Loss: 2.7274
2022-05-22 16:18:43 - Epoch: 99, Validation Loss: 4.2847, Validation Regression Loss 1.3256, Validation Classification Loss: 2.9592
2022-05-22 16:18:44 - Saved model models/amiga/mb1-ssd-Epoch-99-Loss-4.284742280661342.pth
2022-05-22 16:18:44 - Task done, exiting program.
(base) ➔ pytorch-ssd git:(master) X

```

FIGURE A.8: KitchenNet training log

A.2.4 Foreground Detection GMM

In order to model the foreground and background, a Gaussian Mixture Model (GMM) is used. The GMM algorithm is able to learn and produce new pixels distributions based on the data we provided. Therefore, unknown pixels are classified as probable foreground or probable background in light of their colour statistics in relation to the hard-labelled pixels.

Figure A.9 shows some output examples. The test cases consist of both ground and tables.

It is also possible to include foreground detection as part of the secondary image processing node as introduced in Section 6.2.4, which then attempts to segment the foreground from the cropped image. However, the performance is mainly scene-dependent.

Figure A.10 below shows a comparison of GMM to RANSAC in segmenting planes. GMM entails more errors, whereas RANSAC may also extract some point cloud data.

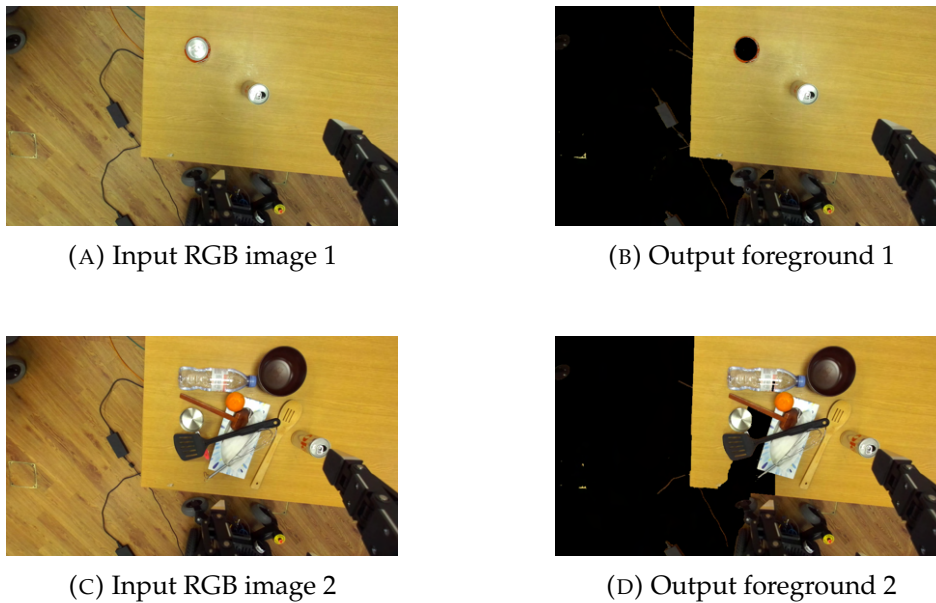


FIGURE A.9: GMM used for foreground detection

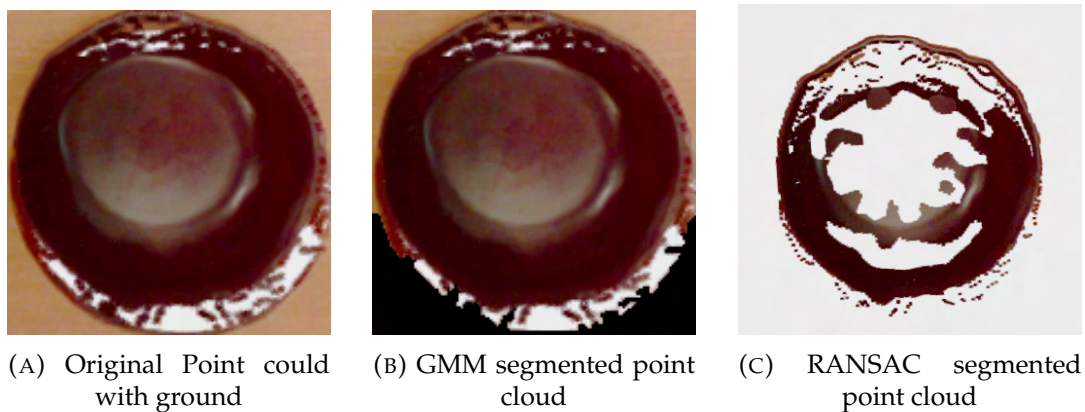


FIGURE A.10: Comparison of GMM to RANSAC in segmenting planes

A.3 Amiga Manipulation

A.3.1 Frame Transforms:

```
@staticmethod
def transform_pose(input_pose, from_frame, to_frame):

    # **Assuming /tf2 topic is being broadcasted
    tf_buffer = tf2_ros.Buffer()
    listener = tf2_ros.TransformListener(tf_buffer)

    pose_stamped = tf2_geometry_msgs.PoseStamped()
    pose_stamped.pose = input_pose
```

```

pose_stamped.header.frame_id = from_frame
pose_stamped.header.stamp = rospy.Time(0)

try:
    # ** It is important to wait for the listener
    # to start listening. Hence the rospy.Duration(1)
    output_pose_stamped = tf_buffer.transform(
        pose_stamped, to_frame, rospy.Duration(2))
    return output_pose_stamped.pose

except (tf2_ros.LookupException,
        tf2_ros.ConnectivityException,
        tf2_ros.ExtrapolationException) as e:
    raise e

@staticmethod
def camera_to_robot(pose_3d : Union[list, Vector3],
                    camera_link : str, robot_link : str='base_link') -> Vector3:
    """
    transform a 3d point from the camera frame to the robot frame
    """
    if isinstance(pose_3d, Vector3):
        pose_3d = [pose_3d.x, pose_3d.y, pose_3d.z]

    listener = tf.TransformListener()
    listener.waitForTransform(robot_link,
                             camera_link, rospy.Time(), rospy.Duration(4.0))
    translation, rotation = listener.lookupTransform(robot_link,
                                                     camera_link, rospy.Time(0))
    cam2rob = np.dot(tf.transformations.
                     translation_matrix(translation),
                     tf.transformations.quaternion_matrix(rotation))
    # build a 3x4 3D affine matrix
    pose_3d_rob = np.dot(cam2rob, np.array(pose_3d+[1]))

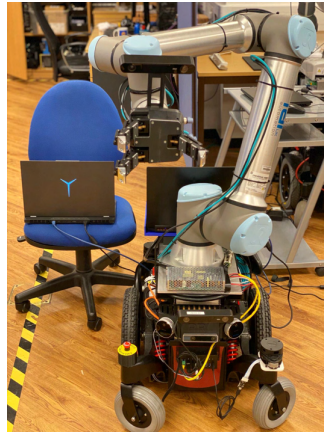
    return Vector3(pose_3d_rob[0], pose_3d_rob[1], pose_3d_rob[2])

```

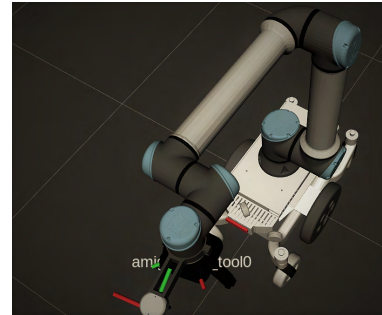
A.3.2 Pre-defined Poses



(A) Navigation safe pose



(B) Grasping front pose



(C) Grasping home pose

FIGURE A.11: Amiga predefined poses

Bibliography

- 3f, Robotiq (2022). *3-Finger Adaptive Robot Gripper*. URL: <https://robotiq.com/products/3-finger-adaptive-robot-gripper>.
- A1M8 (2022). URL: <https://www.slamtec.com/en/Lidar/A1>.
- Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: <https://www.tensorflow.org/>.
- AmazonPickingChallenge Leipzig, Germany (2016). URL: <http://pwurman.org/amazonpickingchallenge/>.
- aruco_ros (2021). URL: https://github.com/pal-robotics/aruco_ros/tree/kinetic-devel.
- autoware_ai_perception (2020). *autoware_ai_perceptio*. URL: https://github.com/autowarefoundation/autoware_ai_perception.
- Ballard, D. H. and C. M. Brown (1982). *Computer Vision*. New Jersey, USA: Prentice Hall. URL: <http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/bandb.htm>.
- Ballardini, Augusto Luis et al. (2014). *ira_laser_tools: a ROS LaserScan manipulation toolbox*. DOI: 10.48550/ARXIV.1411.1086. URL: <https://arxiv.org/abs/1411.1086>.
- Bao, Wentao, Qi Yu, and Yu Kong (2020). *Object-Aware Centroid Voting for Monocular 3D Object Detection*. DOI: 10.48550/ARXIV.2007.09836. URL: <https://arxiv.org/abs/2007.09836>.
- BearRobotics (2022). URL: <https://www.bearrobotics.ai/>.
- Berenson, Dmitry, James Kuffner, and Howie Choset (2008). "An optimization approach to planning for mobile manipulation". In: *2008 IEEE International Conference on Robotics and Automation*, pp. 1187–1192. DOI: 10.1109/ROBOT.2008.4543365.
- Blender (2021). URL: <https://www.blender.org/download/>.
- Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao (2020). *YOLOv4: Optimal Speed and Accuracy of Object Detection*. DOI: 10.48550/ARXIV.2004.10934. URL: <https://arxiv.org/abs/2004.10934>.
- Borkman, Steve et al. (2021). *Unity Perception: Generate Synthetic Data for Computer Vision*. DOI: 10.48550/ARXIV.2107.04259. URL: <https://arxiv.org/abs/2107.04259>.
- Caldera, Shehan, Alexander Rassau, and Douglas Chai (2018). "Review of Deep Learning Methods in Robotic Grasp Detection". In: *Multimodal Technologies and Interaction 2.3*. ISSN: 2414-4088. DOI: 10.3390/mti2030057. URL: <https://www.mdpi.com/2414-4088/2/3/57>.
- CHOMP_Planner (2021). URL: https://ros-planning.github.io/moveit_tutorials/doc/chomp_planner/chomp_planner_tutorial.html.

- Coleman, David et al. (2014). “Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study”. In: *CoRR* abs/1404.3785. arXiv: 1404.3785. URL: <http://arxiv.org/abs/1404.3785>.
- CoppeliaSim (2022). URL: <https://www.coppeliarobotics.com/helpFiles/en/versionInfo.htm#coppeliaSim4.3.0>.
- Corsaro, Matt, Stefanie Tellex, and George Konidaris (2021). *Learning to Detect Multi-Modal Grasps for Dexterous Grasping in Dense Clutter*. arXiv: 2106.03919 [cs.RO].
- Cristianini, Nello and Elisa Ricci (2008). “Support Vector Machines”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, pp. 928–932. ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_415. URL: https://doi.org/10.1007/978-0-387-30162-4_415.
- CUDA (2022). URL: <https://developer.nvidia.com/cuda-downloads>.
- Czarnowski, Jan et al. (2020). “DeepFactors: Real-Time Probabilistic Dense Monocular SLAM”. In: *CoRR* abs/2001.05049. arXiv: 2001.05049. URL: <https://arxiv.org/abs/2001.05049>.
- D435 (2020). URL: https://github.com/pal-robotics/realsense_gazebo_plugin.
- Dalal, N. and B. Triggs (2005). “Histograms of oriented gradients for human detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 1, 886–893 vol. 1. DOI: 10.1109/CVPR.2005.177.
- Damen, Dima et al. (2021a). “Rescaling Egocentric Vision: Collection, Pipeline and Challenges for EPIC-KITCHENS-100”. In: *International Journal of Computer Vision (IJCV)*. URL: <https://doi.org/10.1007/s11263-021-01531-2>.
- Damen, Dima et al. (2021b). “The EPIC-KITCHENS Dataset: Collection, Challenges and Baselines”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 43.11, pp. 4125–4141. DOI: 10.1109/TPAMI.2020.2991965.
- Davision, Andrew (2022). URL: <https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/lecture4.pdf>.
- Davison, Andrew (2022). “Lecture 2: Robot Motion”. In: p. 2. URL: <https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/lecture2.pdf>.
- Deep_Grasp_Demo (2020). URL: https://github.com/PickNikRobotics/deep_grasp_demo.
- Deng, Hao, Jing Xiong, and Zeyang Xia (2017). “Mobile manipulation task simulation using ROS with MoveIt”. In: *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pp. 612–616. DOI: 10.1109/RCAR.2017.8311930.
- Deng, Jia et al. (2009). “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.
- depth_image_proc (2020). URL: http://wiki.ros.org/depth_image_proc.
- depthimage_to_laserscan (2020). URL: http://wiki.ros.org/depthimage_to_laserscan.
- detectNet (2019). URL: <https://github.com/dusty-nv/jetson-inference>.

- Diankov, Rosen (2010). "Automated Construction of Robotic Manipulation Programs". PhD thesis. Carnegie Mellon University, Robotics Institute. URL: http://www.programmingvision.com/rosen_diankov_thesis.pdf.
- Du, Guoguang (2021). URL: <https://github.com/GeorgeDu/vision-based-robotic-grasping#3-2d-planar-grasp/>.
- Du, Guoguang et al. (2020). "Vision-based robotic grasping from object localization, object pose estimation to grasp estimation for parallel grippers: a review". In: *Artificial Intelligence Review* 54.3, pp. 1677–1734. DOI: 10.1007/s10462-020-09888-5. URL: <https://doi.org/10.1007%2Fs10462-020-09888-5>.
- dynamic_reconfigure (2015). *ROS_dynamic_reconfigure*. URL: http://wiki.ros.org/dynamic_reconfigure.
- easy_handeye (2021). URL: https://github.com/IFL-CAMP/easy_handeye.
- find_object_2d (2016). URL: http://wiki.ros.org/find_object_2d.
- Fischler, M. and R. Bolles (1981). "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Communications of the ACM* 24.6, pp. 381–395. URL: <http://publication.wilsonwong.me/load.php?id=233282275>.
- Fox, Dieter et al. (Jan. 1999). "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots". In: pp. 343–349.
- Gazebo (2022). URL: <https://gazebo-sim.org/home>.
- Ge, Zheng et al. (2021). *YOLOX: Exceeding YOLO Series in 2021*. DOI: 10.48550/ARXIV.2107.08430. URL: <https://arxiv.org/abs/2107.08430>.
- Girshick, Ross (2015). *Fast R-CNN*. DOI: 10.48550/ARXIV.1504.08083. URL: <https://arxiv.org/abs/1504.08083>.
- Girshick, Ross et al. (2013). *Rich feature hierarchies for accurate object detection and semantic segmentation*. DOI: 10.48550/ARXIV.1311.2524. URL: <https://arxiv.org/abs/1311.2524>.
- gmapping (2019). URL: <http://wiki.ros.org/gmapping>.
- Grasp_fix (2021). URL: <https://github-wiki-see.page/m/JenniferBuehler/gazebo-pkgs/wiki/The-Gazebo-grasp-fix-plugin>.
- Grasp_Gazebo (2018). URL: <https://github.com/jsbruglie/grasp>.
- Hackathon, IROS (2018). "TIAGo robot multitasking at the IROS Mobile Manipulation Hackathon". In: URL: <https://blog.pal-robotics.com/tiago-robot-iros-mobile-manipulation-hackathon/>.
- Hartley, Richard and Andrew Zisserman (2004). "Bibliography". In: *Multiple View Geometry in Computer Vision*. 2nd ed. Cambridge University Press, 634–645. DOI: 10.1017/CB09780511811685.037.
- Hauser, Kris (2020). URL: <https://motion.cs.illinois.edu/RoboticSystems/InverseKinematics.html>.
- Hawkins, Kelsey P. (2013). URL: https://smartech.gatech.edu/bitstream/handle/1853/50782/ur_kin_tech_report_1.pdf.
- He, Kaiming et al. (2014). "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *Computer Vision – ECCV 2014*. Springer International Publishing, pp. 346–361. DOI: 10.1007/978-3-319-10578-9_23. URL: https://doi.org/10.1007%2F978-3-319-10578-9_23.

- He, Kaiming et al. (2017). *Mask R-CNN*. DOI: 10.48550/ARXIV.1703.06870. URL: <https://arxiv.org/abs/1703.06870>.
- Hess, Wolfgang et al. (2016). “Real-time loop closure in 2D LIDAR SLAM”. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1271–1278. DOI: 10.1109/ICRA.2016.7487258.
- Heydarzadeh, Mehrdad et al. (Feb. 2017). “Remaining Useful Lifetime Estimation for Power MOSFETs under Thermal Stress with RANSAC Outlier Removal”. In: *IEEE Transactions on Industrial Informatics PP*. DOI: 10.1109/TII.2017.2665668.
- Hornung, Armin et al. (2013). “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees”. In: *Autonomous Robots*. Software available at <https://octomap.github.io>. DOI: 10.1007/s10514-012-9321-0. URL: <https://octomap.github.io>.
- Huang, Jonathan et al. (2016). *Speed/accuracy trade-offs for modern convolutional object detectors*. DOI: 10.48550/ARXIV.1611.10012. URL: <https://arxiv.org/abs/1611.10012>.
- image_geometry (2009). URL: http://docs.ros.org/en/kinetic/api/image_geometry/html/python/index.html.
- Inertia, List of moments of (2022). URL: https://en.wikipedia.org/wiki/List_of_moments_of_inertia.
- ira_laser_tools (2021). URL: https://github.com/iralabdisco/ira_laser_tools.
- Itseez (2015). *Open Source Computer Vision Library*. <https://github.com/itseez/opencv>.
- Jiang, Yun, Stephen Moseson, and Ashutosh Saxena (2011). “Efficient grasping from RGBD images: Learning using a new rectangle representation”. In: *2011 IEEE International Conference on Robotics and Automation*, pp. 3304–3311. DOI: 10.1109/ICRA.2011.5980145.
- Joystick, R-NET (2022). URL: <https://www.cw-industrialgroup.com/Products/Mobility-Vehicle-Solutions/R-net/Joystick-Modules-LED>.
- Kalakrishnan, M. et al. (2011). “STOMP: Stochastic trajectory optimization for motion planning”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. clmc. Shanghai, China, May 9-13. URL: <http://www-clmc.usc.edu/publications/K/kalakrishnan-ICRA2011.pdf>.
- Kinematics_and_Dynamics_Library (2021). *Kinematics_and_Dynamics_Library*. URL: https://github.com/orocos/orocos_kinematics_dynamics.
- kinova (2020). URL: <https://www.kinovarobotics.com/product/gen3-robots>.
- Krasin, Ivan et al. (2017). “OpenImages: A public dataset for large-scale multi-label and multi-class image classification.” In: URL: <https://storage.googleapis.com/openimages/web/index.html>.
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. URL: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>.

- Kuznetsova, Alina et al. (2020). "The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale". In: *IJCV*. URL: <https://arxiv.org/abs/1811.00982>.
- Kümmerle, Rainer et al. (2011). "G2o: A general framework for graph optimization". In: *2011 IEEE International Conference on Robotics and Automation*, pp. 3607–3613. DOI: [10.1109/ICRA.2011.5979949](https://doi.org/10.1109/ICRA.2011.5979949).
- L515, Intel REALSENSE (2022). URL: <https://www.intelrealsense.com/lidar-camera-l515/>.
- Lang, Alex H. et al. (2018). *PointPillars: Fast Encoders for Object Detection from Point Clouds*. DOI: [10.48550/ARXIV.1812.05784](https://doi.org/10.48550/ARXIV.1812.05784). URL: <https://arxiv.org/abs/1812.05784>.
- laser_filter (2021). URL: http://wiki.ros.org/laser_filters#scan_to_scan_filter_chain.
- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *nature* 521.7553, p. 436.
- Lepetit, Vincent, Francesc Moreno-Noguer, and Pascal Fua (Feb. 2009). "EPnP: An accurate O(n) solution to the PnP problem". In: *International Journal of Computer Vision* 81. DOI: [10.1007/s11263-008-0152-6](https://doi.org/10.1007/s11263-008-0152-6).
- Levine, Sergey et al. (2016). *Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection*. DOI: [10.48550/ARXIV.1603.02199](https://doi.org/10.48550/ARXIV.1603.02199). URL: <https://arxiv.org/abs/1603.02199>.
- Liang, Ming et al. (2019). "Multi-Task Multi-Sensor Fusion for 3D Object Detection". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Lin, Tsung-Yi et al. (2014). *Microsoft COCO: Common Objects in Context*. DOI: [10.48550/ARXIV.1405.0312](https://doi.org/10.48550/ARXIV.1405.0312). URL: <https://arxiv.org/abs/1405.0312>.
- Lin, Tsung-Yi et al. (2016). *Feature Pyramid Networks for Object Detection*. DOI: [10.48550/ARXIV.1612.03144](https://doi.org/10.48550/ARXIV.1612.03144). URL: <https://arxiv.org/abs/1612.03144>.
- Liu, Guanfeng, J.C. (Jeff) Trinkle, and Richard Milgram (Jan. 2004). "Complete Path Planning for a Planar 2-R Manipulator with Point Obstacles." In: vol. 4, pp. 3263–3269. DOI: [10.1109/ROBOT.2004.1308757](https://doi.org/10.1109/ROBOT.2004.1308757).
- Liu, Wei et al. (2016). "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Springer International Publishing, pp. 21–37. DOI: [10.1007/978-3-319-46448-0_2](https://doi.org/10.1007/978-3-319-46448-0_2). URL: https://doi.org/10.1007/978-3-319-46448-0_2.
- Lowe, David G. (Nov. 2004). "Distinctive Image Features from Scale-Invariant Keypoints". In: *Int. J. Comput. Vision* 60.2, pp. 91–110. ISSN: 0920-5691. DOI: [10.1023/B:VISI.0000029664.99615.94](https://doi.org/10.1023/B:VISI.0000029664.99615.94). URL: <http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94>.
- Lynch, Kevin M. and Frank C. Park (2017). *Modern Robotics: Mechanics, Planning, and Control*. 1st. USA: Cambridge University Press. ISBN: 1107156300.
- Mahler, Jeffrey et al. (2016). "Dex-net 1.0: A cloud-based network of 3d objects for robust grasp planning using a multi-armed bandit model with correlated rewards". In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1957–1964.

- Mahler, Jeffrey et al. (2017). “Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics”. In: Mahler, Jeffrey et al. (2019). “Learning ambidextrous robot grasping policies”. In: *Science Robotics* 4.26, eaau4984.
- MAKRSHAKR (2022). URL: <https://www.makrshakr.com/>.
- Merkel, Dirk (2014). “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239, p. 2.
- MeshLab (2022). URL: <https://www.meshlab.net/>.
- Mikolajczyk, Krystian (2021). *Lecture 4, CNN architectures*.
— (2022). *Lecture 3, Computer Vision and Pattern Recognition*.
- Milenkovic, Victor, Elisha Sacks, and Steven Trac (2013). “Robust Complete Path Planning in the Plane”. In: *Algorithmic Foundations of Robotics X*. Ed. by Emilio Frazzoli et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 37–52. ISBN: 978-3-642-36279-8.
- MiR1350 (2021). “The most powerful AMR from MiR to date”. In: MisoRobotics (2022). URL: <https://misorobotics.com/flippy-2/>.
- Mitash, Chaitanya, Abdeslam Boularias, and Kostas E. Bekris (2017). *Improving 6D Pose Estimation of Objects in Clutter via Physics-aware Monte Carlo Tree Search*. DOI: 10.48550/ARXIV.1710.08577. URL: <https://arxiv.org/abs/1710.08577>.
- Moley (2022). URL: <https://moley.com/>.
- Mousavian, Arsalan, Clemens Eppner, and Dieter Fox (2019). *6-DOF Grasp-Net: Variational Grasp Generation for Object Manipulation*. DOI: 10.48550/ARXIV.1905.10520. URL: <https://arxiv.org/abs/1905.10520>.
- MoveIt_Calibration (2020). URL: https://github.com/ros-planning/moveit_calibration.
- MoveIt_Grasps (2020). URL: https://github.com/ros-planning/moveit_grasps.
- MoveIt_Interface (2022). URL: http://docs.ros.org/en/noetic/api/moveit_ros_planning_interface/html/classmoveit_1_1planning_interface_1_1MoveGroupInterface.html.
- MoveIt_Task_Constructor (2022). URL: https://github.com/ros-planning/moveit_task_constructor.
- Mur-Artal, Raul, J. M. M. Montiel, and Juan D. Tardós (2015). “ORB-SLAM: a Versatile and Accurate Monocular SLAM System”. In: *CoRR* abs/1502.956. arXiv: 1502.00956. URL: <http://arxiv.org/abs/1502.00956>.
- Newcombe, Richard A. et al. (2011). “KinectFusion: Real-time dense surface mapping and tracking”. In: *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 127–136. DOI: 10.1109/ISMAR.2011.6092378.
- Nvidia (2019). URL: <https://rawgit.com/dusty-nv/jetson-inference/dev/docs/html/python/jetson.inference.html#detectNet>.
- NX, XAVIER (2020). URL: <https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-xavier-nx/>.
- Object_Detection_Model_from_EPIC-KITCHENS (2020). URL: <https://data.bris.ac.uk/data/dataset/gw5q146a9dqd2t82jhu8noe3m>.
- ode (2004). URL: <https://www.ode.org/>.

- OpenCV (2000). URL: https://docs.opencv.org/3.4/d9/d0c/group_calib3d.html.
- opengl (1992). URL: <https://www.opengl.org/>.
- openvino (2022). URL: <https://github.com/openvinotoolkit/openvino>.
- Pan, Jia, Sachin Chitta, and Dinesh Manocha (2012). "FCL: A general purpose library for collision and proximity queries". In: *2012 IEEE International Conference on Robotics and Automation*, pp. 3859–3866. DOI: [10.1109/ICRA.2012.6225337](https://doi.org/10.1109/ICRA.2012.6225337).
- Pas, Andreas ten et al. (2017). *Grasp Pose Detection in Point Clouds*. DOI: [10.48550/ARXIV.1706.09911](https://doi.org/10.48550/ARXIV.1706.09911). URL: <https://arxiv.org/abs/1706.09911>.
- Pas Marcus Gualtieri, Kate Saenko Andreas ten and Robert Platt (2019). URL: <https://github.com/atenpas/gpd>.
- Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems* 32. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- Pepper, SoftBank Robotics (2014). "A robot designed to interact with humans". In: URL: <https://www.softbankrobotics.com/emea/en/pepper>.
- Pi, Raspberry (2019). URL: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>.
- Pinto, Lerrel and Abhinav Gupta (2015). *Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours*. DOI: [10.48550/ARXIV.1509.06825](https://doi.org/10.48550/ARXIV.1509.06825). URL: <https://arxiv.org/abs/1509.06825>.
- pointcloud_to_laserscan (2015). URL: http://wiki.ros.org/pointcloud_to_laserscan.
- Power, R-NET (2022). URL: <https://www.cw-industrialgroup.com/Products/Mobility-Vehicle-Solutions/R-net/EL-Power-Module>.
- pythonPCL (2018). URL: <https://github.com/strawlab/python-pcl>.
- Qi, Charles R et al. (2017). "Frustum PointNets for 3D Object Detection from RGB-D Data". In: *arXiv preprint arXiv:1711.08488*.
- Qin, Zengyi, Jinglu Wang, and Yan Lu (2019). "MonoGRNet: A Geometric Reasoning Network for 3D Object Localization". In: *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*.
- Quickie (2022). URL: <https://www.sunrisemedical.co.uk/wheelchairs/quickie/lightweight-wheelchairs>.
- R-Net_System (2017). URL: <https://www.cw-industrialgroup.com/Products/Mobility-Vehicle-Solutions>.
- Ratliff, Nathan D. et al. (2009). "CHOMP: Gradient optimization techniques for efficient motion planning". In: *2009 IEEE International Conference on Robotics and Automation*, pp. 489–494.
- Realsense_ROS (2018). URL: <https://github.com/IntelRealSense/realsense-ros>.
- Redmon, Joseph and Anelia Angelova (2014). *Real-Time Grasp Detection Using Convolutional Neural Networks*. DOI: [10.48550/ARXIV.1412.3128](https://doi.org/10.48550/ARXIV.1412.3128). URL: <https://arxiv.org/abs/1412.3128>.

- Redmon, Joseph and Ali Farhadi (2016). *YOLO9000: Better, Faster, Stronger*. DOI: 10.48550/ARXIV.1612.08242. URL: <https://arxiv.org/abs/1612.08242>.
- (2018). *YOLOv3: An Incremental Improvement*. DOI: 10.48550/ARXIV.1804.02767. URL: <https://arxiv.org/abs/1804.02767>.
- Redmon, Joseph et al. (2015). *You Only Look Once: Unified, Real-Time Object Detection*. DOI: 10.48550/ARXIV.1506.02640. URL: <https://arxiv.org/abs/1506.02640>.
- Ren, Shaoqing et al. (2015). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. DOI: 10.48550/ARXIV.1506.01497. URL: <https://arxiv.org/abs/1506.01497>.
- Reynolds, Douglas (2009). “Gaussian Mixture Models”. In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil Jain. Boston, MA: Springer US, pp. 659–663. ISBN: 978-0-387-73003-5. DOI: 10.1007/978-0-387-73003-5_196. URL: https://doi.org/10.1007/978-0-387-73003-5_196.
- Roa, Maximo A. et al. (2021). “Mobile Manipulation Hackathon: Moving into Real World Applications”. In: *IEEE Robotics Automation Magazine* 28.2, pp. 112–124. DOI: 10.1109/MRA.2021.3061951. URL: <https://ieeexplore.ieee.org/document/9380912>.
- RoboCup (2019). “TIAGo at RoboCup 2019 podium”. In: URL: <https://blog.pal-robotics.com/tiago-robot-robocup-2019/>.
- RobotIq_Driver (2018). URL: <https://github.com/ros-industrial/robotiq>.
- robotiq_github (2019).
- RobotIq_User_Manual (2019). URL: https://assets.robotiq.com/website-assets/support_documents/document/3-Finger_PDF_20190221.pdf.
- ROS (2021). URL: <https://www.ros.org/>.
- ros_base_local_planner (2014). URL: http://wiki.ros.org/base_local_planner.
- ROSbot_2.0 (2022). URL: https://github.com/husarion/rosbot_ros.
- ROS_Calib (2020). URL: http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration.
- ROS_Control (2013). URL: https://classic.gazebo-sim.org/tutorials?tut=ros_control.
- ROS_CV_Bridge (2010). URL: http://wiki.ros.org/cv_bridge.
- rosDebug (2017). URL: <http://wiki.ros.org/tf/Debugging%20tools>.
- ros_deep_learning (2022). URL: https://github.com/dusty-nv/ros_deep_learning.
- ros_ekf (2021). URL: https://github.com/ros-planning/robot_pose_ekf.
- roslaunch (2019). URL: <http://wiki.ros.org/roslaunch>.
- ros_navfn (2014). URL: <http://wiki.ros.org/navfn>.
- ROS_Navigation_Stack (2022). URL: <https://github.com/ros-planning/navigation>.
- ros_nodelet (2022). URL: <http://wiki.ros.org/nodelet>.
- ROS_Speech (2021). *speech*. URL: <http://wiki.ros.org/speech>.
- rplidar_ros (2022). URL: https://github.com/Slamtec/rplidar_ros.
- rqt, ROS (2016). URL: <http://wiki.ros.org/rqt>.
- RTAB_Map (2022). URL: <http://introlab.github.io/rtabmap/>.

- rtabmap_odometry (2021). URL: http://wiki.ros.org/rtabmap_ros#rgbd_odometry.
- Rusu, Radu Bogdan and Steve Cousins (2011). "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE.
- Rviz, ROS (2019). URL: <http://wiki.ros.org/rviz>.
- Salisbury, Prof. Kenneth (2010). "Develop platform technology for research and development where robots do mobile manipulation tasks in human environments". In: URL: <http://personalrobotics.stanford.edu/>.
- Saranya, C. et al. (2014). "Real Time Evaluation of Grid Based Path Planning Algorithms: A comparative study". In: *IFAC Proceedings Volumes 47.1*. 3rd International Conference on Advances in Control and Optimization of Dynamical Systems (2014), pp. 766–772. ISSN: 1474-6670. DOI: <https://doi.org/10.3182/20140313-3-IN-3024.00050>. URL: <https://www.sciencedirect.com/science/article/pii/S1474667016327410>.
- Schapire, Robert E (2013). "Explaining adaboost". In: *Empirical inference*. International Springer Publisher, pp. 37–52.
- Shi, Shaoshuai, Xiaogang Wang, and Hongsheng Li (2018). *PointRCNN: 3D Object Proposal Generation and Detection from Point Cloud*. DOI: [10.48550/ARXIV.1812.04244](https://arxiv.org/abs/1812.04244). URL: <https://arxiv.org/abs/1812.04244>.
- Shiu, Y.C. and S. Ahmad (1989). "Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form $AX=XB$ ". In: *IEEE Transactions on Robotics and Automation* 5.1, pp. 16–29. DOI: [10.1109/70.88014](https://doi.org/10.1109/70.88014).
- Simos, Theodore E., Vasilios N. Katsikis, and Spyridon D. Mourtas (2022). "Multi-input bio-inspired weights and structure determination neuronet with applications in European Central Bank publications". In: *Mathematics and Computers in Simulation (MATCOM)* 193.C, pp. 451–465. DOI: [10.1016/j.matcom.2021.11..](https://ideas.repec.org/a/eee/matcom/v193y2022icp451-465.html) URL: <https://ideas.repec.org/a/eee/matcom/v193y2022icp451-465.html>.
- Spot, Boston Dynamics (2016). "four-legged canine-inspired Spot". In: URL: <https://www.bostondynamics.com/products/spot>.
- statistical_outlier_removal (2020). URL: http://www.open3d.org/docs/latest/tutorial/Advanced/pointcloud_outlier_removal.html.
- Şucan, Ioan A., Mark Moll, and Lydia E. Kavraki (2012). "The Open Motion Planning Library". In: *IEEE Robotics & Automation Magazine* 19.4. <https://ompl.kavrakilab.org>, pp. 72–82. DOI: [10.1109/MRA.2012.2205651](https://doi.org/10.1109/MRA.2012.2205651).
- Sundermeyer, Martin et al. (2021). *Contact-GraspNet: Efficient 6-DoF Grasp Generation in Cluttered Scenes*. DOI: [10.48550/ARXIV.2103.14127](https://arxiv.org/abs/2103.14127). URL: <https://arxiv.org/abs/2103.14127>.
- SWIFTI, ABB (2020). "SWIFTI: Industrial speeds, collaborative skills". In: URL: <https://cobots.robotics.abb.com/en/robots/swifti>.
- T265, Intel REALSENSE (2022). URL: <https://www.intelrealsense.com/tracking-camera-t265/>.
- Tedrake, Russ (2021). *Robotic Manipulation. Perception, Planning, and Control*. URL: <http://manipulation.mit.edu>.

- TensorRT (2022). URL: <https://docs.nvidia.com/deeplearning/tensorrt/index.html>.
- tf, ROS (2017). URL: <http://wiki.ros.org/tf>.
- TF, ROS (2020). URL: <http://wiki.ros.org/tf/Tutorials>.
- TRAC Labs_Robotics (2021). URL: https://bitbucket.org/traclabs/trac_ik/src/master/.
- Tremblay, Jonathan et al. (2018). "Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects". In: *Conference on Robot Learning (CoRL)*. URL: <https://arxiv.org/abs/1809.10790>.
- UR10e (2022). URL: <https://www.universal-robots.com/products/ur10-robot/>.
- urdf, ROS (2012). URL: <http://wiki.ros.org/urdf/XML/model>.
- UR_Industrial (2021). URL: https://github.com/ros-industrial/universal_robot.
- UR_ROS_Driver (2021). *Universal_Robotics_ROS_Driver*. URL: https://github.com/UniversalRobots/Universal_Robots_ROS_Driver.
- Viola, P. and M. Jones (2001). "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1, pp. I-I. DOI: [10.1109/CVPR.2001.990517](https://doi.org/10.1109/CVPR.2001.990517).
- Wang, Cong et al. (2020). "Learning Mobile Manipulation through Deep Reinforcement Learning". In: *Sensors* 20.3. ISSN: 1424-8220. DOI: [10.3390/s20030939](https://doi.org/10.3390/s20030939). URL: <https://www.mdpi.com/1424-8220/20/3/939>.
- xacro (2022). URL: <https://github.com/ros/xacro>.
- XL-GEN, Robotnik (2016). "The mobile manipulator XL-GEN arises from the union among Kinova Robotics and Robotnik featuring an excellent platform". In: URL: <https://robotnik.eu/products/mobile-manipulators/xl-gen/>.
- yoloROS (2021). URL: https://github.com/leggedrobotics/darknet_ros.
- ZED2 (2022). URL: <https://www.stereolabs.com/zed-2/>.
- ZED2_ObjectDetect (2022). URL: <https://www.stereolabs.com/docs/object-detection/>.
- ZED_ROS (2022). URL: <https://github.com/stereolabs/zed-ros-wrapper>.
- Zeng, Andy et al. (2016). *Multi-view Self-supervised Deep Learning for 6D Pose Estimation in the Amazon Picking Challenge*. DOI: [10.48550/ARXIV.1609.09475](https://doi.org/10.48550/ARXIV.1609.09475). URL: <https://arxiv.org/abs/1609.09475>.
- Zhang, Hanbo et al. (2018). *ROI-based Robotic Grasp Detection for Object Overlapping Scenes*. DOI: [10.48550/ARXIV.1808.10313](https://doi.org/10.48550/ARXIV.1808.10313). URL: <https://arxiv.org/abs/1808.10313>.
- Zhao, Zhong-Qiu et al. (2018). *Object Detection with Deep Learning: A Review*. DOI: [10.48550/ARXIV.1807.05511](https://doi.org/10.48550/ARXIV.1807.05511). URL: <https://arxiv.org/abs/1807.05511>.
- Zhou, Qian-Yi, Jaesik Park, and Vladlen Koltun (2018). "Open3D: A Modern Library for 3D Data Processing". In: *arXiv:1801.09847*.
- Zhou, Xinwen et al. (2018). *Fully Convolutional Grasp Detection Network with Oriented Anchor Box*. DOI: [10.48550/ARXIV.1803.02209](https://doi.org/10.48550/ARXIV.1803.02209). URL: <https://arxiv.org/abs/1803.02209>.

Zhu, Xingkui et al. (2021). *TPH-YOLOv5: Improved YOLOv5 Based on Transformer Prediction Head for Object Detection on Drone-captured Scenarios*. DOI: [10.48550/ARXIV.2108.11539](https://arxiv.org/abs/2108.11539). URL: <https://arxiv.org/abs/2108.11539>.