Universität Bremen
Faculty 3, Mathematics and Computer Science

Bachelorthesis

Robots learning geometric groundings of object arrangements for household tasks from virtual reality demonstrations

Ableiten von geometrischen Objektpositionierungen für Haushaltsaufgaben durch Maschinelles Lernen mithilfe von Virtuelle Realität Daten

for the purpose of obtaining the degree Bachelor of Science

Author: Thomas Lipps <tlipps@uni-bremen.de>
MatNr. 4346238

Version from: April 18, 2020

1. Supervisor: Prof. Dr. Michael Beetz
2. Supervisor: Dr. René Weller
Advisor: Gayane Kazhoyan

Universität Bremen

Offizielle Erklärungen von

Nachname: Lipps Vorname: Thomas

Matrikelnr.: 4346238

A) Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.
Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht.
Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein.
Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach u. Jahr.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☒ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

C) Einverständniserklärung über die Bereitstellung und Nutzung der Bachelorarbeit / Masterarbeit / Hausarbeit in elektronischer Form zur Überprüfung durch Plagiatssoftware

Eingereichte Arbeiten können mit der Software *Plagscan* auf einen hauseigenen Server auf Übereinstimmung mit externen Quellen und der institutionseigenen Datenbank untersucht werden.
Zum Zweck des Abgleichs mit zukünftig zu überprüfenden Studien- und Prüfungsarbeiten kann die Arbeit dauerhaft in der institutionseigenen Datenbank der Universität Bremen gespeichert werden.

☒ Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum Zweck der Überprüfung auf Plagiate auf den *Plagscan*-Server der Universität Bremen hochgeladen wird.

☐ Ich bin ebenfalls damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck auf dem *Plagscan*-Server der Universität Bremen hochgeladen u. dauerhaft auf dem *Plagscan*-Server gespeichert wird.

☐ Ich bin nicht damit einverstanden, dass die von mir vorgelegte u. verfasste Arbeit zum o.g. Zweck auf dem *Plagscan*-Server der Universität Bremen hochgeladen u. dauerhaft gespeichert wird.

Mit meiner Unterschrift versichere ich, dass ich die oben stehenden Erklärungen gelesen und verstanden habe. Mit meiner Unterschrift bestätige ich die Richtigkeit der oben gemachten Angaben.

15.04.2020, Bremen

Datum, Ort

Unterschrift

Abstract

Learning object placements for different tasks is essential for robots acting in human households, to perform their actions robustly and flexibly in different simulated and real-world environments. Since the placement of objects depends highly on the environment and the human user using the kitchen, the robot can execute actions by imitating the different humans in the particular environment. For this the robot observes the human storing and placing objects in different scenarios. Virtual Reality allows with complex simulations to create realistic environments for humans executing tasks. A human could set e. g. a table for breakfast in the virtual environment and collect at the same time data which can be interpreted by robots. The robot can use the different object positions and orientations during the breakfast setting, to learn models representing different object placements and relations. By using the learned data the robot can efficiently execute high level manipulation actions and thus setting a table for breakfast.

Zusammenfassung

Das Erlernen von Objektpositionierungen aus unterschiedlichen Aufgaben is essentiel für Küchenroboter, um flexibel und realistisch in unterschiedlichen Umgebungen zu arbeiten. Da die Objektplatzierungen abhängig sind von der Küche und den Menschen die sie nutzen, führt der Roboter Aktionen so aus wie es die unterschiedlichen Menschen in der gegebenen Küche gemacht haben. Dafür beoachtet der Roboter den Menschen wie er oder sie die Objekte verstaut oder in unterschiedlichen Szenarien platziert. Virtual Reality erlaubt mit komplexen Simulationen realisitische Umgebungen für Menschen zu kreieren, in denen unterschiedliche Aktionen wie in der echten Küche ausgeführt werden können. Z. B. könnte ein Mensch in der virtuellen Umgebung ein Tisch für das Frühstück vorbereiten und würde gleichzeitig dabei Daten aufnehmen, die vom Roboter interpretiert werden können. Der Roboter kann die unterschiedlichen Objektpositionierungen während des Tischdeckens nutzen, um ein Modell zu erlernen, welches die unterschiedlichen Objektplatzierungen repräsentiert. Mit den gelernten Daten kann der Roboter dann effizient komplexe Manipulationsaufgaben ausführen und somit einen Tisch für das Frühstück decken.

Contents

1 Introduction **6**
1.1 Motivation . . . 6
1.2 Hypothesis . . . 8
1.3 Scope of this Thesis . . . 8
1.4 Contribution . . . 8
1.5 Structure of this Thesis . . . 9

2 Related Work **10**

3 Foundations **12**
3.1 KnowRob . . . 12
3.1.1 Interface . . . 12
3.2 VR . . . 12
3.3 ROS . . . 13
3.4 CRAM . . . 15
3.4.1 Prolog . . . 15
3.4.2 Designators . . . 16
3.4.3 Process Modules . . . 18
3.4.4 Location-Costmap . . . 18
3.5 Bullet Simulation . . . 20
3.6 VR Data Pipeline . . . 21
3.7 Python Packages . . . 22

4 Approach **24**
4.1 Pipeline . . . 24
4.2 Acquisition of Data . . . 24
4.2.1 RobCoG and the Unreal Engine . . . 24
4.2.2 Querying of KnowRob . . . 29
4.3 Object Placement Learning Model . . . 32
4.3.1 Assumptions . . . 32
4.3.2 Architecture . . . 32
4.3.3 Implementation . . . 37

5 Evaluation **49**

6 Conclusion **54**
6.1 Summary . . . 54
6.2 Discussion . . . 54
6.3 Future Work . . . 57

Bibliography **58**

Appendix **60**
Figures . . . 60
List of Figures . . . 68
List of Tables . . . 70
Listings . . . 71
Acronyms . . . 72

1 Introduction

1.1 Motivation

Developments in robot hardware started in the recent years to get more efficient, robust and usable to build systems which can be operated and work in the real world. The flexibility in the design of state of the art robots led to robots with great natural locomotion and to robots with fast and precise execution of tasks, which are efficiently used in factories. But still these need to be directed and often do not represent their environment and therefore are used as tools. If the robot needs to work with the environment around it or with other humans, the needed information for executing a task increases greatly. Giving a human and robot the same task shows how much implicit knowledge is hidden behind a simple task. In this bachelor thesis, the given domain is not a factory, but a kitchen. A kitchen robot can be deployed to assist the human in the kitchen by cooking or setting up the table for him or her. The tasks in a kitchen are highly complex since knowledge must be utilized for cooking meals or even cleaning dishes. Let us assume, that e. g. the kitchen robot should setup a table for breakfast. While the human starts with the task, the robot may already fail because it does not know how to grasp or place objects reasonably. Even if particular objects are handed to the robot and it is acceptable to just drop these instead of placing carefully, questions like “Where are the wanted objects stored? “, “On which surface should these be placed?“ or “Where exactly should the robot place the cups or plates on the given surface?“ rise. What for the robot is needed knowledge, is for the human commonsense. Thus, we either save static information about the environment on the robot specifying e. g. placements of spoons and plates or we try to learn these placements. Since the former method results in little flexibility, the latter should be realized for real-world table settings in an arbitrary user’s home.

To learn a specific task like a table setting, a model is required which is able to return the missing information. Its definition must be understandable for the robot and the structure must allow adjustments. With adjusting the model the robot may be able to learn from observations or other data, thus needed information gets returned allowing to solve the problem of the task. Models and algorithms which are able to learn from specific data are available from the research field of Artificial Intelligence (AI). Tasks like the estimation of the robot position, state and the perceiving of its environment can be solved in robotics with various AI models and methods. These learn and acknowledge different types of information, allowing the execution of a specific tasks after using the learned model. How the task is specified depends on the domain the robot is working in. The built model in this bachelor thesis allows the robot to set tables for breakfast by placing objects on surfaces like the human did. It imitates the human by learning and saving where objects were taken from and how they were arranged. Moreover, it will be

able to allow to represent relations between objects, so the robot can like the human orientate itself on the already placed objects. Thus, if a number of objects are already set on the table, the robot will be able to find positions for other objects, relative to the already placed ones.

But how exactly does the robot acquire the commonsense of the human which can be represented by a defined model? Due to advancements in Virtual Reality (VR), the opportunity rises to use the humans commonsense and task knowledge encoded in virtual actions. After the robot watched a human setting up a table in VR, it is able to tell where the plate should be placed and that e. g. the spoon should be on the right side of the plate. Moreover, the robot will be able to do that for more than one human, if the human showed it how. Therefore, the robot will examine the data collected from the human to complete tasks like setting a table for breakfast. The goal of this thesis is to make it possible for the robot to ground placements for different object types by examining the human demonstration. In addition, the robot should distinguish between the breakfast setups of different humans too.

In the field of robotics, two different approaches are common in capturing the motions of objects and the manipulation of the environment: either the experiment is recorded by a video camera or it takes place in VR and is recorded by a computer. Both of these approaches are sufficient enough to conclude semantic or specific information about the objects and environment. In this thesis, the VR approach will be used.

The main reason is, that VR allows much more flexibility and is easier for configuring the environment. Kitchen environments can be changed easily in a simulation by e. g. moving furniture or importing different items. Although VR does not perceives the real world, modern computers allow with complex and realistic simulation environments, the deployment of a VR setup delivering photo-realistic images. Moreover, VR allows to record more conveniently everything changing in VR. Videos, on the other hand, must first recognize, what they are filming and can only recognize what is in the field of view of the camera. Whereas in VR one has access to the complete world, independent of the camera position or orientation.

Furthermore, a knowledge-based system called KnowRob allows to access the VR data with a convenient interface, thus only needed data can be saved and used for this bachelor thesis. With VR, the data must only be recorded and exported, so that the knowledge-based system can filter the information in the VR data. Hence, the knowledge-based system allows to comfortably export filtered data, it can be used directly to train a model representing the symbolic or concrete placements of different objects.

The planning framework CRAM [19] makes it together with the learned model possible to execute a simulation of a robot setting a table for a breakfast scenario using different objects and their placings.

1.2 Hypothesis

This bachelor thesis uses a imitation learning approach to learn symbolic and subsymbolic object placements for table setups in the given kitchen. Since the placements are dependent from the human the robot learned from, the built system allows different humans, table, kitchens and table settings (e. g. lunch, dinner). The robot can therefore ask in any state of the kitchen where to place objects and how to orientate them. The built system answers with available placements since already placed objects on the table are recognized. This means in particular that the sequence of placing objects does not matter and that the built system returns relational placements which lead to more suitable breakfast table settings.

The collected data gets fitted in a chosen machine learning model, which represents the placements of the different object types on the table. To validate this, the outputs of the model are visualized and discussed. Moreover, the system was connected via a ROS interface with the planner framework CRAM to check its suitability and usability in a simulation environment.

1.3 Scope of this Thesis

The target of this bachelor thesis is not a continuous system, which infers in real-time new placements of objects or adjusts the behavioral model of the human. Therefore, the model in the built system will be initialized once and returns then the same learned placements. It is not possible to add new data from VR experiments dynamically while the system runs and it does not learn from queries coming from the planning framework CRAM [19]. This does not mean that the system is not able to recognize redundant information. If e. g. bowls are already placed on the table, the system recognizes this and returns placement information accordingly. Moreover, the system prefers relational placements between objects. So if a spoon should be placed on the table the system recognizes the already placed bowls and returns placements being in relation to the placed objects. Therefore, stacking of objects or other redundant errors will be prevented on the planning level and commonsense of the human will be used to execute the given tasks successfully. Finally, this bachelor thesis will not be applied on the real PR2 robot, but only on the simulated PR2 robot in the simulation Bullet [5].

1.4 Contribution

The contribution part of this bachelor thesis contains mainly two parts. First, the query functions heading towards the knowledge-based system KnowRob [2] were expanded. This was done to export the collected data in the VR experiments in a CSV-file. This file is used in the built system to represent the kitchen and its objects and to learn parameters to conclude the placements of used objects.

The second part contains a ROS [21] interface to communicate between the planning framework CRAM [19] and the built system, since the planning framework is written in Common Lisp and the built system in Python. Furthermore, more functions in CRAM were implemented and used in the reasoning components of CRAM too, so the learned placements could be used fluently in the planning environment. Finally, and most importantly, a model was designed and implemented to represent suitable object placements in relation to other objects, by using Gaussian Mixture Models (GMM).

This thesis presents a fully integrated pipeline, where data can be acquired in VR, then filtered through KnowRob queries into a CSV-file, after which a machine learning model is trained, which can then be queried by the robot with the CRAM framework to give locations where to search for objects and where to place them on the table.

1.5 Structure of this Thesis

Chapter 2: Related Work gives a insight in the state of the art developments for planning complex tasks and learning of object placements and relations.

Chapter 3: Foundations introduces the used frameworks and explains them each and shows how they are used together.

Chapter 4: Approach describes the built model by starting with its input parameters and its specification. Afterwards, the used model is explained by visualizing it and discussed theoretically to show its suitability for the desired tasks explained in the Hypothesis.

Chapter 5: Evaluation shows the built model in practice, validates it and proves its usability and practical suitability for the desired tasks explained in the Hypothesis.

Chapter 6: Conclusion summarizes this bachelor thesis, discusses the capabilities of the built model and presents further improvements for the future.

2 Related Work

Executing complex high-level plans like a table setting for breakfast contain many parameters. Parameters like the objects most likely location or its position and orientation goal, are filled from humans through their commonsense and experience. Instead of using humans commonsense robots can be simulated and execute actions in a loop to specialize for a task in a given environment. In [18] fetching and placing tasks of objects are specialized by being validated in a simulated closed loop. Therefore, concrete values for the above parameters are learned to successfully apply these on the real robot to fetch and pick up objects. Although this approach does create more stable pick and place actions, objects may not be arranged sensible enough for table settings.

The general approach of teaching robots trough imitation is popular in robotics ([23], [7]). To accomplish that two widespread and common technologies exist for acquiring data from humans: either by filming the human doing the task or by putting the human in VR where the task should be completed in. In [25] researchers showed after performing a task under the observation of the robot using a RGB camera, that the robot could mimic this specific task successfully. This was based on the reconstruction of the hands and objects trajectories. The matching hand and object poses were defined as a graph problem and solved by a graph optimization library. In [10] similar was done but with collected VR data instead. The VR data was transferred in the knowledge system KnowRob to reason on it. The results constructed trajectories of used objects in different scenarios. In a pancake making scenario the spatula and pancake trajectories could be calculated. In [13] the pancake scenario is referred again in the context of establishing a failure detection model, which was trained from a human in a physics simulation.

Besides the applications of imitation learning in trajectory following and failure detection, commonsense is been used in different parameters of action planning too ([12], [16]). In [24] researchers built a generative model, which learned - after fitting it with videos showing everyday activities - placements of different objects and causal dependencies between actions. The filmed objects are arranged by representing their placements through GMMs [9]. The causal dependencies between objects and physical contexts like “glass on this table“ are learned from a Recurrent Neural Network (RNN) [6] which can predict future actions too. The results show generated new manipulation animations from objects, action predictions and motion planning. Other researchers used different methods to learn object placements or relations between objects and the manipulator. In [3] Support Vector Machines (SVM) [4] are used instead of GMMs to cluster data points, which is sufficient enough to represent object placements on a surface. To represent simple “on“ and “adjacent“ relations between objects Rosman and Ramamoorthy developed an algorithm in [22] and applied it on different pictures showing correctly classified relationships.

Using videos as input results in different challenges having to be solved. First, the objects must be reliably recognized as well as the actions performed from the human. Moreover, the modeling of the human activity and character animation are difficult tasks due to the complexity of human movements [24]. These challenges are more convenient to solve by using VR to collect data. In [1] researches constructed a graph representing related activities in the virtual environment by acquiring VR data and extract semantic information on which was reasoned on. The result contains actions related to objects like grasping, reaching, taking, staying idle or simple relations of objects like “GlassOnSponge“. Because of the on-going research in the field of knowledge-based reasoning in complex everday activites, sufficient results in from of table settings arrangements ([12], [15]) were achieved. In [17] an application of imitation learning was applied by collecting data in VR and using the knowledge-based system KnowRob [2] to reason on the collected data. Moreover, it gives a great insight in the inner implementation of fetching and placing actions [16] in CRAM [19]. To successfully fetch an object it is vastly important for the robot to know where to stand for registering the object, where to place its end-effector and how to grasp the object (e.g. from the top, behind). This information is recorded in a VR setup and was saved in a database inside of the knowledge-based system KnowRob [2] from where it can be accessed trough queries written in first-order (FO) logic. After the analysis of the plan code, the needed motion parameters were summarized and a probabilistic model was build. This model utilizes a Fuzzy Markov Logic Network, which returns the probability of success for fetching an object given the input of a robot position, robot arm, object orientation and on which side the object was laid on. The experiments in table setting context showed that the initial random pose of objects in the simulation had a big impact for the success of fetching it. Moreover, the results for the tests on the real robot concluded, that not much VR data is needed for ensuring successful motion parameterization.

3 Foundations

In this chapter the software components that have previously existed and have been applied in this thesis for the querying and transporting of information collected in VR are described.

3.1 KnowRob

One prerequisite for constructing a system like in this bachelor thesis intended is a knowledge system, which could reason on the collected VR data. KnowRob [15] is a knowledge processing system that combines knowledge representation and reasoning. The knowledge is represented in the Web Ontology Language (OWL). Reasoning is done by referencing the stored knowledge in Prolog as presented in Subsection 3.4.1.

The knowledge processing system is used in this bachelor thesis to reason on the VR data and extract crucial information. It contains knowledge about the world in which the robot moves and how to execute actions with objects in the world to achieve a wanted goal. Therefore, it can answer questions from the planning level like: when did the human grasp something? Where was his hand as he started grasping it? Where was he looking while doing this? Which hand was used and what was grasped? Where was his hand located and how was the orientation before he grabbed? When did the grasping start and when did it end? These questions need to be answered to make the robot execute complex activities like e. g. a table setting. Since knowledge about the objects and environment are necessary to make a complex task feasible, the planning framework CRAM and KnowRob need to be connected via an interface.

3.1.1 Interface

The package cram_json_prolog[1] is used as an interface to create an question-answer-system from CRAM towards KnowRob. The ROS json prolog client was implemented in CRAM to allow the use of json-prolog within Common Lisp. Therefore, prolog queries can be sent in a JSON format via ROS to KnowRob. The results can then be used in the following plan execution in CRAM.

3.2 VR

To collect data in VR, one needs a realistic environment. In my case I needed data for table settings, so the environment is a realistic kitchen. In [11] is described how the VR simulation was designed to build an as realistic as possible kitchen environment. The built kitchen in VR is simulated in the Unreal Engine[2] allowing naive physics like dynamic pushing and gravitation forces on objects. Therefore, drawers can be opened

[1] cram_json_prolog: `https://github.com/cram2/cram/tree/boxy/cram_json_prolog`
[2] Unreal Engine: `https://www.unrealengine.com/en-US/`

and closed, objects can be moved by picking or placing, or they can bump at each other.

All the manipulation activities the human performs in the virtual environment and the effects of performing in the virtual environment are recorded. This includes the poses of hand, camera, object and kitchen objects. While the human performs tasks in VR, positions and orientations of all objects are sent to a database. Together with the time-synchronized events, a hierarchical symbolic-subsymbolic activity representation gets created. The symbolic part is defined in FO logic with rules like shown in Listing 1.

```
1         ep_inst(EpInst),
2         obj_type(ObjInst, knowrob:'cup'),
3         obj_type(EventInst, knowrob:'GraspingSomething'),
4         u_occurs(EpInst, EventInst, Start, End)
```

Listing 1: Base information to query VR data in KnowRob with Prolog

Every VR session will be saved as one episode. Episodes are organized in events like TouchingSomething or GraspingSomething. `EpInst` is an unbound episode variable in the call of `ep_inst()`, so Prolog can choose one episode instance from the collected VR data pool. Similarly in line two of Listing 1 an object instance `ObjInst` of given object type `cup` and in line three an event instance `EventInst` of type `GraspingSomething` get queried. Finally, Prolog checks in line four if the queried event instance `EventInst` is in the episode instance `EpInst` within the two timestamps `Start` and `End`. Since we got the timestamps for a specific episode and object instance we can get the positions and orientations for this object instance from `Start` to `End`. Therefore, the knowledge base KnowRob, which saves the symbolic data, is used to query subsymbolic data like trajectories of objects, kitchen parts or human hands on a symbolic level.

For making the above feasible, the symbolic knowledge base in KnowRob represents all objects in VR. Moreover, the subsymbolic data is segmented in motions like “GraspingSomething“ and categorized to actions modeled as a sequence of motions [8].

The package RobCoG[3] exports the data in JSON- and OWL-files. These JSON-files contain different events like grasping and touching of objects, states of links in the kitchen and opening and closing of kitchen drawers or doors. Every recording session in VR creates a JSON-file. The OWL-file contains the semantic map of the VR kitchen.

3.3 ROS

The Robot Operating System (ROS) [21] allows for an efficient, productive and fail-safe work environment with robots. ROS is not an operating system (OS), because it operates on top of an OS like Linux. ROS software is categorized mainly in two parts.

[3] RobCoG: `https://github.com/robcog-iai/RobCoG`

The first part called “main“ contains general tools making it possible to compute the robot system on distributed systems. These systems need to interact with each other to establish a working robot system. Therefore, they are connected via a communication interface and need to be built independently from each other with a package and build tool. The ROS build system is catkin[4], which is a CMake extension being able to specify how to build, test and deploy the developed systems. The “main“ package is developed by companies and external developers.

The second part of ROS is called “universe“ and is developed by the open ROS community. It contains various packages like the transform library tf[5] or the computer vision library opencv[6] and algorithms like the Inverse Kinematic Calculation from the Kinematics and Dynamics Library kdl[7]. These are used to achieve the execution of autonomous robotic tasks and avoid reinventing the wheel. Moreover, the ROS community offers hardware drivers, visualization tools like rviz[8] and different graphical user interfaces (GUI) for analyzing data sent via the ROS communication interface.

The ROS communication layer (of ROS version 1) includes in practice the nodes, which each represent one of the distributed systems, and the roscore. Nodes can be considered as nodes in a graph. Each node runs processes and consumes and/or produces data via typed topics. Therefore, data can be published or collected by subscribing to specific topics of other nodes. With that concept, ROS allows great flexibility allowing asynchronous many-to-many data streams. But ROS hands an ability to retrieve filtered information too, by providing typed services. Each service depends on the node it is running on. It offers a synchronous communication channel, which answers incoming typed data by directly responding with other typed data and, thus, allows for that reason to question other nodes for specific filtered information. ROS provides different command line tools to visualize and access specific information from the running ROS network. rqtgraph[9] visualizes the nodes connected through topics or services. rosnode[10] allows getting information about the running nodes and the advertised topics. rostopic[11] delivers information about the topics and adds an interface to interact with all topics by sending typed data through the command line interface.

All data sent via the ROS communication layer is typed by message (msg) or service (srv) files. These message or service files can be written from developers by using the base standard types from ROS. Since ROS supports different programming languages, these must be translated into classes of the supported programming languages allowing to utilize them inside the developers’ implementation.

[4]catkin: http://wiki.ros.org/catkin
[5]tf: http://wiki.ros.org/tf
[6]opencv: http://wiki.ros.org/vision_opencv
[7]kdl: http://wiki.ros.org/kdl
[8]rviz: http://wiki.ros.org/rviz
[9]rqtgraph: http://wiki.ros.org/rqt_graph
[10]rosnode: http://wiki.ros.org/rosnode
[11]rostopic: http://wiki.ros.org/rostopic

The roscore needs to be started to create a ROS communication network. It holds the ROS master, a parameter server and rosout. The ROS Master is a XML-RPC server with the primary target to connect different nodes, e. g., through topics. Before a node starts to advertise a topic, it registers the topic at the ROS master. If one node wants to publish some data on this topic, it asks the ROS master on which node the topic is running. The ROS master provides the wanted node and the connection between both nodes gets initiated. The parameter server holds the configuration files of the nodes and rosout is a network-based stdout to log time-sensitive information like warnings or errors processed on different nodes.

3.4 CRAM

The Cognitive Robotic Abstract Machine (CRAM) [19] is a task planning framework controlling autonomous robots in a real or simulated environment. It allows with its geometric grounding and fast simulation methods for the construction of high-level robot control plans. Furthermore, it can reason about the past task executions and optimize its plans for better performance.

CRAM defines prewritten basic action plans like picking, perceive, navigating and complex ones like fetching, delivering and transporting. Since CRAM allows to use different robots, plans for these have to be general and modular. Moreover, in robotics it is important that tasks are easy embeddable, transparent and at least interruptible.

Due to these general requirements, plans in CRAM are written in the CRAM Plan Language (CPL). CPL offers fluents and entity descriptions, which allow basic control structures during runtime. Fluents represent the state of the robot by defining variables, which can be thread-safe manipulated. But since planning actions does not fit in classical programming planning, these fluents are not sufficient enough to plan a specific task. They are just powerful enough to verify with primitive statements if a specific goal for an action was achieved. The description represents actions and the environment abstractly in symbols and need to be referenced to infer missing information about the action or environment. The reactive and dynamic referencing approach with Prolog allows to plan stereotypical actions enough to execute them in a simulation or on the real robot. To give a greater insight in CRAM, the different types of descriptions are explained in Subsection 3.4.2.

3.4.1 Prolog

Prolog is a declarative and logic programming language in which it is possible to define facts and rules to check if a particular clause is true. A rule in pure Prolog is a implication from a conjunction of clauses to one clause. The conjunction of clauses is called body and the single clause is called head. A rule is true, if the body is true too.

A fact is just a rule without body. An example can be found in the appendix in Figure 25.

CRAM has its own primitive Prolog interpreter written in Common Lisp[12] allowing to use Common Lisp designator objects (see Subsection 3.4.2) and other data structures in Prolog variables. In CRAM it is possible to define facts and rules inside of fact-groups. Therefore, designators can be resolved by e. g. validating it in the body of a specified rule with prolog rules and facts. If assignments to the prolog variables exist, such that all elements of the body are true, the head of the rule gets evaluated with the assignments of the variables. Furthermore, Prolog can answer queries by returning concrete values for given parameters. If some assignments for a rule or fact in the body fails or returns NIL, Prolog tries with another assignment. If one rule or fact in the body cannot be true, the designator resolving fails too meaning that the head will not be executed. Finally, in CRAM the Prolog interface returns a lazy list. This allows together with the defined referencing rules in the fact groups dynamic inferring of designators and an opportunity for a reactive implementation of CRAM plans. An example can be found in the appendix in Figure 26.

3.4.2 Designators

In CRAM the symbolic entity description is an attribute of a Common Lisp designator class. The syntactic definition of the description is a key-value-pair list. Each designator must be resolved. They are resolved by validating the values in the description and inferring the missing keys and values by using Prolog rules. Therefore, the resolved designator changes and the description gets extended with more information included. Resolved designators have a solution bound to them and are called effective designators. The saving of the designator changes allows reasoning about the past. This is done by equating the unresolved and effective designator object. This means that a new designator object will be created for the effective designator with an updated timestep and description. Moreover, the parent slot of the resolved designator object points at the unresolved designator object. The unresolved designator object gets therefore an updated successor entry, saving the pointer towards the effective designator object.

3.4.2.1 Object Designator

Object designators describe objects like e. g. a cup by specifying its type, color or name. Furthermore, they can describe kitchen furniture like drawers or different surfaces.

Referencing The typical use of object designators is the detection of objects by a perception system. The resolution of object designators requires therefore, that the robot is already in a position in which it is able to detect the object with its perception

[12] cram_prolog: `https://github.com/cram2/cram/tree/master/cram_core/cram_prolog`

system. If the perception system is a camera in the head of robot, at least the rotation of the head must be set to be able to detect the object.

3.4.2.2 Motion Designator

Motion designators are responsible for the low-level motions the robot should complete. These motions are then forwarded to the hardware-dependent process-modules.

Referencing Motion designators are atomic motions including movements for the robots base, torso or arm(s), and commands for the end effector or perception system for detecting objects. Each of these motions are defined in a group or alone to specific process module handling the specified motions in the real world or simulation. Since motion designators are very basic, the referencing needs more concrete information meaning to define and execute motion designators manually more explicit knowledge must be filled into the description of the designator.

3.4.2.3 Location Designator

Location designators describe locations of objects, the robot or parts of the environment and can return distributions representing e. g. from which positions objects are reachable or visible.

Referencing Location designators are not referenced in Prolog, but through a sampling-based approach with location-generator and location-validation functions. The referenced location designators return at the end a lazy list of coordinates in different frames.

First the generator functions take the unresolved location designator and collect lazy lists of possible solutions. The generator functions are specified with a priority value, to specifiy the sequence in which the generator functions should be called. Therefore, the function with the highest priority gets called first. After all generation functions were called, a solution is verified by a sequence of validation functions by sampling from the generated lazy lists. This means the validation function gets as input the location designator and one generated solution. A solution will be discarded immediately, if one validation function rejected the solution. If at least one validation function accepted the solution and the others returned UNKOWN, the solution is accepted. A validation function returns ACCEPT if the solution is valid or returns REJECT if it is invalid. Moreover, if the validation function cannot decide, UNKNOWN will be returned. Finally a validation function can return MAYBE-REJECT too, meaning if the solution will not be accepted by any other validation function then the solution will be rejected.

Since the validators and generators need to be specified from the developer, in CRAM the generators and validators are implemented with the Location-Costmaps class.

3.4.2.4 Action Designator

Action designators describe high-level actions which need more motions or interactions with the environment to perform its desired task.

Referencing Since it is possible to structure cognitive actions in a sequence of different actions [8], CRAM defines the actions as a executable hierarchical structure [16]. In CRAM this is implemented with different types of action designators. If we want e. g. to deliver an object the robot picked up, the robot must first navigate to a position, then turn and look towards the position where the picked object should be placed. Each of these actions has its own parameters ([8], [16]). Therefore, for each action an action designator is defined with a specific action type. Every action designator will then be resolved with the use of knowledge systems and CRAM Prolog to infer missing paramteres. One big problem of this structuring is, that the sequence of actions builds up an dependency of actions: e. g. the robot navigated to a position where it cannot place an object safely. This problem is solved in CRAM by catching errors thrown e. g. during the placing process and then resampling of loction designators with location-costmaps (see Subsection 3.4.4) of the e. g. navigation poses. Each action sequence ends with execution of motion designators.

3.4.3 Process Modules

Process modules are used as an interface towards the hardware dependent subsystems of the robot. The basic approach in defining these is: they get a motion designator, they resolve the motion designator and pass the parameters of the designator to a function, sending the commands e. g. towards the movement system of the real robot. Since it does not matter for the task planning how the robot moves towards the desired target these hardware dependent actions can be excluded from the high-level robot control program. This abstraction layer allows together with the modularization of different process modules for more flexibility since the integration of other robots or hardware systems needs only adaptions in the process modules.

3.4.4 Location-Costmap

Location designators describe locations in symbolic ways. Listing 2 defines in a location designator placements for an object of object type bowl on the dining table. Therefore, every pose on the dining table, which is suitable for the bowl, is represented by this designator.

At runtime, this designator needs to be resolved into a specific pose in the robots environment. As there are multiple, even an infinite number of poses, that satisfy the symbolic description, the Location-Costmap mechanism of CRAM represents areas of

```
1 (a location
2     (for bowl)
3     (on (an object
4         (type dining-table))))
```

Listing 2: Example of a location designator

space, that satisfy these constraints. Additionally, Location-Costmaps allow to sample randomly from these areas, to get one sample pose.

Location-Costmap is in CRAM implemented as a Common Lisp class containing meta-information about the map, the map itself saved as a 2D-matrix and three lists of functions named cost-functions, height-generators and orientation-generators. The meta-information of the map contains information such as the reference point of the map, the height and width of the map and the resolution. The map represents a distribution in the by the meta-information specified area and the functions in cost-functions, height-generators and orientation-generators are used to calculate the cost value, orientation and height for different coordinates of the map.

The integration of Location-Costmap objects into the resolving process of a location designator is done by registering the generator and validation functions of the Location-Costmap as generators and validation functions of the location designators.

Resolving a location designator in CRAM starts therefore with the execution of the location costmap generator function, which samples from the map of the Location-Costmap object. If the map was not initialized yet, the cost-functions of the Location-Costmap object create a new map by calculating a value in $[0, 1]$ for every entry in the 2D-matrix. Therefore, the cost values are being used as a probability value for each point in the distribution. Points with a higher probability or cost value are more likely to get sampled. After the distribution in map was calculated, one entry in map gets sampled. With the sampled entry and the meta-information, it is possible to calculate the corresponding coordinate. The coordinate gets passed to the height- and orientation-generator functions of the Location-Costmap object, to calculate the height and orientation of the given coordinate. The sampled coordinate with the calculated height and orientation are then encoded as a pose. To allow resampling the sample is returned in a lazy-list.

The location-costmap-pose-validator function gets therefore as input the location-designator object and the generated pose, which should be validated.

Since every entry in the matrix map was calculated and created a distribution of points, this can be visualized in the bullet simulation. Figure 1 shows an example of the Location-Costmap objects in practice. This costmap visualizes poses for the robot to stand to perceive the bowl. The height-generators of this costmap return a Z coordinate of 0, as these are poses for the robot to stand on the floor. The orientation values are not represented visually until the robot moves to the position and orientates

itself accordingly as shown in Figure 2. The cost values are represented in the height of the squares and in the colors from blue to red, represented as a heat map.

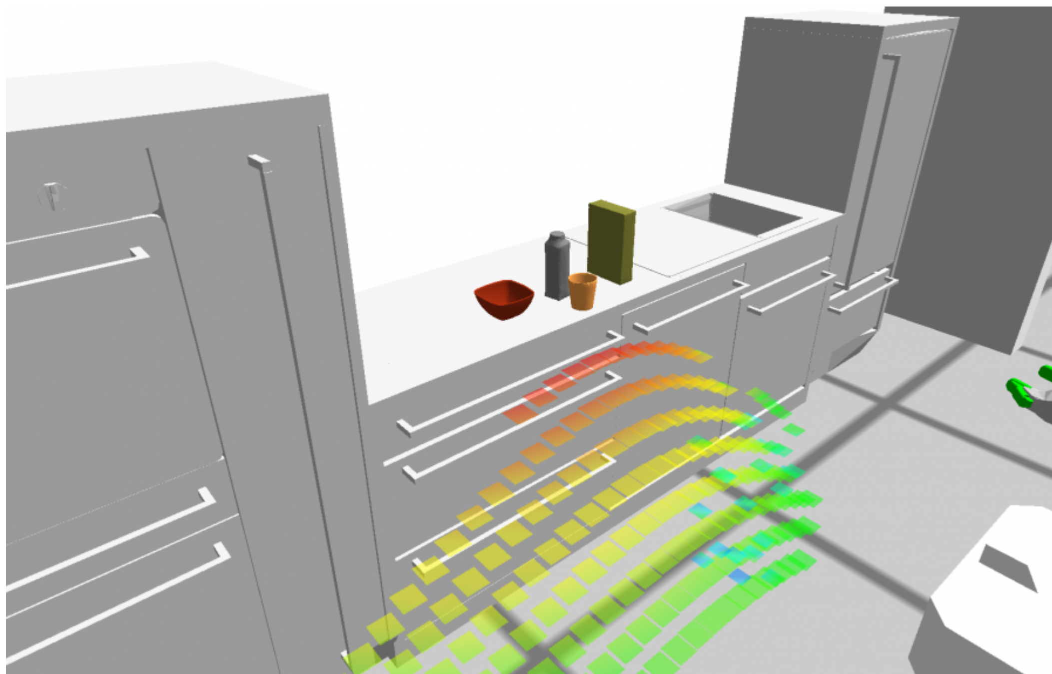


Figure 1: Visibility costmap of the bowl represented as Gaussian distribution

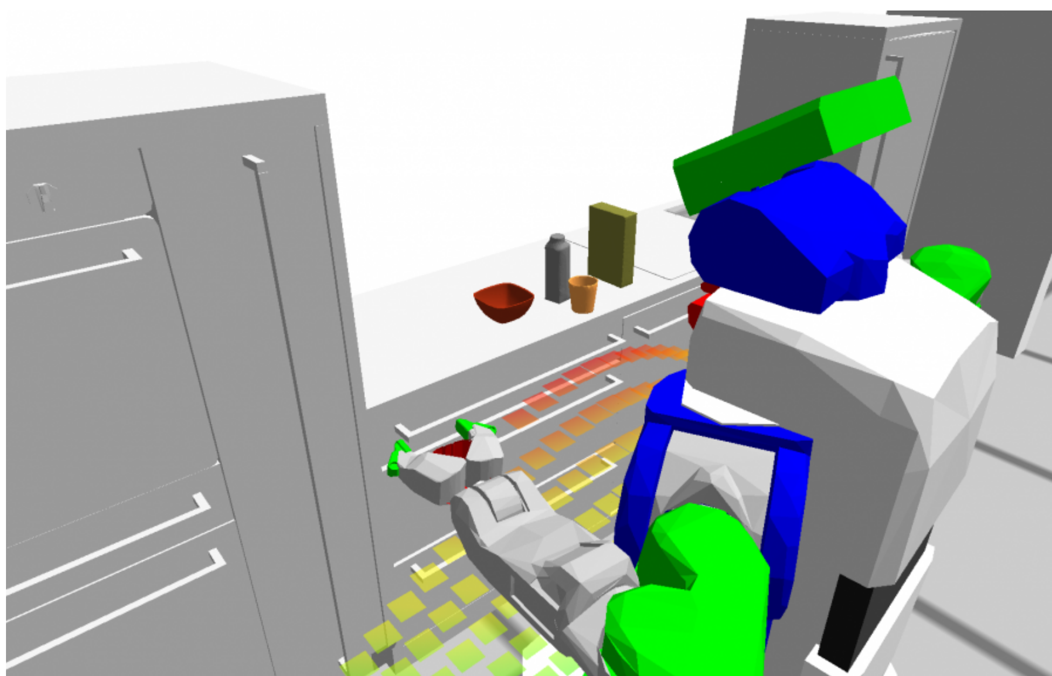


Figure 2: Visibility costmap of bowl represented as Gaussian distribution showing the orientation of the generated/sampled and validated pose

3.5 Bullet Simulation

Bullet [5] is a free and lightweight physics engine used in CRAM (see Figures 1, 2), which is able to simulate collision detection and dynamics of objects. It was specifically

chosen for CRAM since it allows abstracting away low-level movements and has a basic physics simulation for e. g. the gravitation. In the planning scope of CRAM it does not matter how e. g. a robot moves from one position to another. Therefore, the simulation from one position to another is cut out, which saves enormously resources during the execution of the plans in CRAM. Generally speaking, every movement or detection on Process Module level is abstracted in the simulation by jumping directly to the goal. Since Bullet is able to simulate collision detection, in CRAM an error is thrown, if the robot e. g. tries to move to the center of the table or wants to put its end-effector inside the table. Due to the simulation, plans can be optimized through a trial and error approach to infer parameters which could execute the plan on the real robot.

3.6 VR Data Pipeline

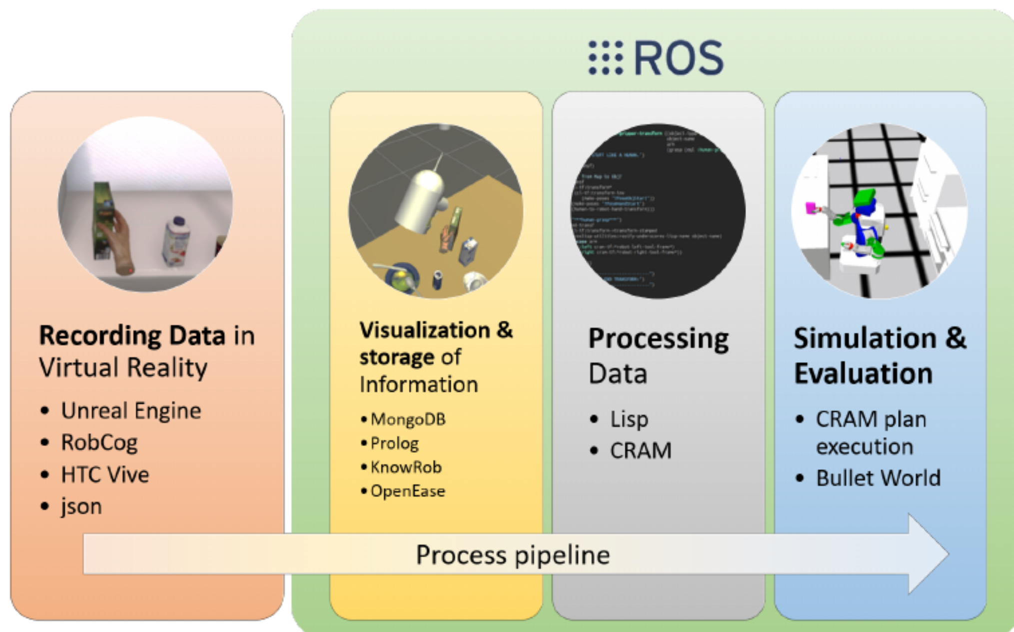


Figure 3: The pipline from the Unreal Engine until the robot simulation, Figure taken from [14]

In [14] a pipeline was built to allow reasoning and planning with the collected VR data. Figure 3 shows an abstract view of the pipeline processing the VR data. It starts with the recording of data in the virtual environment. For this a HTC Vive setup was used. The collected VR data is then exported with RobCoG (see Section 3.2). For executing tasks including multiple different action plans, different plan-specific parameters need to be filled by the planning tool CRAM [16]. Some of theses parameters can be filled with information gathered in VR episodes. Therefore, all the VR data was imported into KnowRob. This was done by manually inserting the subsymbolic data in the data base

MongoDB from where it can be accessed by the KnowRob addon knowrob_robcog[13]. This addon was specifically developed to reason with KnowRob on the collected VR data. The connection of the knowledge-based system KnowRob and the planning framework CRAM was established by using ROS and the package cram_json_prolog (see Subsection 3.1.1). The package cram_json_prolog allows to send Prolog queries from CRAM which are evaluated with knowrob_robcog in KnowRob on the collected VR data. Since the plan execution needs specific parameters filled with fitting values, different Prolog queries were written in the CRAM package cram_knowrob_vr[14] to access the information for specific planning tasks. The referencing of different plans in the planner framework CRAM can therefore access specific information collected in VR experiments. Once the action could be resolved successfully, it is simulated in the simulation Bullet.

3.7 Python Packages

Since the system that I built to learn poses for the table setting scenario was implemented in Python, different Python libraries were used to establish a model in the time frame of a bachelor thesis being feasible enough to handle the requested tasks.

Data Handling pandas was used to import the created CSV-file, since it has countless operations allowing to filter and categorize comfortably for specific data.

List and matrix operations numpy was used for different list and matrix operations allowing fast calculations. Moreover, numpy is utilizes by the other used libraries too.

Visualization matplotlib was used together with scipy and seaborn to visualize the fitted placement models.

Learning sklearn was used since it has well documented implementation of GMMs and various other models. Moreover, it contained operations for metrics which allowed for evaluation and debugging of the used models.

Caching diskcache was used to cache the learned object placements. Therefore, these did not have to be initialized again after every restart.

[13]knowrob_robcog: `https://github.com/robcog-iai/knowrob_robcog`
[14]cram_knowrob_vr:`https://github.com/cram2/cram/tree/boxy/cram_knowrob/cram_knowrob_vr`

Communication The ROS library rospy was used to start a ROS node running different ROS services. Moreover, it offered operations to log errors and info statements on the network level.

4 Approach

In this chapter the acquiring of the VR data and the built model which learned form the collected VR data are explained and presented.

4.1 Pipeline

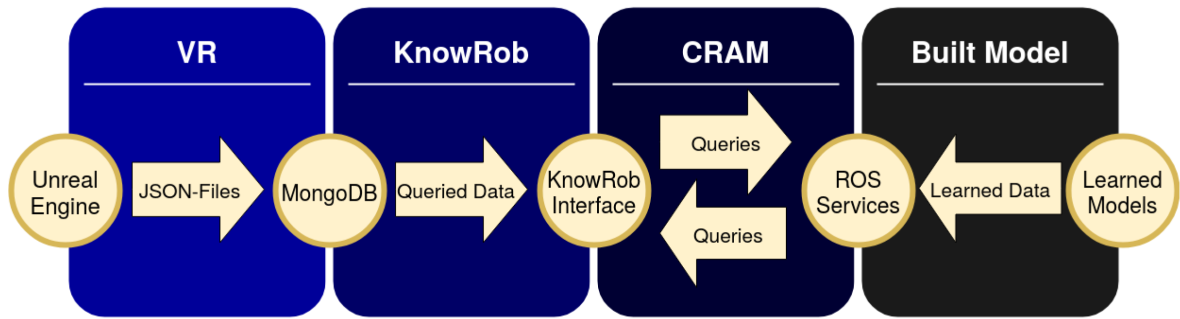


Figure 4: The data flow of the symbolic and subsymbolic placement information acquired from VR experiments

Figure 4 shows the data flow of the symbolic and subsymbolic data collected in VR experiments. The pipeline starts with the Unreal Engine in VR. First the human starts acquiring data by performing VR experiments. In VR, I collected VR data by setting the table with various objects for breakfast as explained in Subsection 4.2.1. The different VR experiments were then exported into JSON-files, which were imported into the database MongoDB as explained in Section 3.6. KnowRob can access the exported VR data in the MongoDB and allows with its interface to answer queries by reasoning on the acquired VR data (see Section 3.1). After the data was loaded successfully into MongoDB, queries from CRAM (see Section 3.2) allow to access the symbolic and subsymbolic placements of the VR objects from KnowRob. All the queried information was then exported in a CSV-file. The contents of the CSV-file are described in Subsection 4.2.2. The built system described in Section 4.3, imports the CSV-file and learns the placements of the used VR objects by fitting a machine learning model described in Subsection 4.3.3.1. After the built system was initialized, it allows with its ROS services to be queried by CRAM. Therefore, the simulated robot in CRAM can access the learned placements of various objects during planning and executing of pick and place tasks to set the table for breakfast like the human did in VR.

4.2 Acquisition of Data

4.2.1 RobCoG and the Unreal Engine

First, the data was collected. For this a HTC Vive Set was used including VR glasses and a joystick for each hand. With a trigger on each of these, a hand in VR could

be closed and opened to grasp objects in VR. The package RobCoG and the Unreal Engine (see Section 3.2) made it possible to collect VR data comfortably with the given hardware. The simulation in the Unreal Engine contained the same kitchen and its furniture as in the simulation Bullet in CRAM. Thus, the furniture measurements fit to these in the simulation in Bullet. Furthermore, the Unreal Engine allowed to export the kitchen as a semantic map saved in a OWL-file, which allowed to load the kitchen from the Unreal Engine in the Bullet simulation too.

Figure 5: Simulated kitchen environment showing on the left the **kitchen island** with drawers and on the right the **kitchen sink area**. Left from the **sink area** is some space on the work place and in the left corner is an oven coated from two **pull-out shelves**. The right **pull-out shelf** is opened. The **fridge** is on the right of the **kitchen sink area**. Moreover, this Figure shows a full breakfast setup on the **kitchen island**.

With every start of a VR experiment a JSON-file was created. In this file are the trajectories of the arms, used objects and camera saved. Thus, it includes the symbolic and subsymbolic information of the object poses too. The object poses at the start of picking actions and at the end of placing actions were the most interesting for fulfilling the task described in the Hypothesis 1.2. RobCoG implemented these start and end poses by triggering the events GraspingSomething and PlacingSomthing whenever an object was picked or placed anywhere in the kitchen. Therefore, if the end pose of an object instance should be queried, the PlacingSomething event allowed to distinguish the timestamp at the end of the action, so that the object instances pose in the VR kitchen could be retrieved. Additionally, the symbolic information contained links in the kitchen which were manipulated by e. g. opening a drawer or closing the fridge door.

An example event timeline is presented in Figure 6 representing the manipulation of the kitchen and an object in VR. It shows that initially the link SinkDrawerLeft-Middle of the kitchen supported the object Cup_80jZ. During the experiment the SinkDrawerLeftMiddle-Link was manipulated and the cup object was grasped with the right hand and short after that supported by the IslandArea-Link in the kitchen. This means that a cup was grasped with the right hand out of a drawer, which was before opened. Then the grasped cup was placed on the table with the surface called IslandArea. At the end, the opened drawer was closed again.

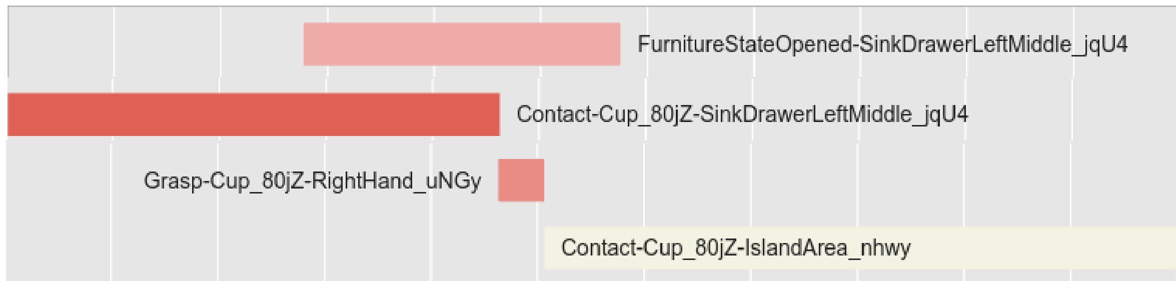


Figure 6: The timeline of the VR experiment showing the manipulation of a cup and the kitchen

To accomplish as many placement poses as possible in each VR session and collect as many data points as possible, the table was set with different plates, bowls, cutlery, cups, mugs and drinks. The kitchen setup in the Unreal Engine shown in Figure 5 was copied from the real kitchen environment shown in Figure 7. Moreover, the kitchen already stored different objects as shown in Figure 8 to e. g. set the table for breakfast for five or more people. The kitchen stored small and big glasses, cups and mugs in the lower drawers of the kitchen sink area as shown in Figure 5 and in the upper drawer cutlery like forks, knives and spoons as shown in Figure 27. In the drawers of the kitchen island different plates and bowls were hidden, which are visualized in Figure 5 too. Moreover, the fridge contained milk and orange juice. Lastly, the right pull-out shelf in the kitchen contained different cereal types as presented in Figure 5 which were used too.

Figure 7: Real kitchen environment showing on the left the **kitchen island** with drawers and on the right the **kitchen sink area**. Left from the **sink area** is some space on the work place and in the left corner is an oven coated from two **pull-out shelves**. The **fridge** is on the right of the **kitchen sink area**. The photo shows the robot PR2 doing a pick and place task on the **kitchen island**. Location: Laboratory of the Institute of Artificial Intelligence in the University Bremen. Figure taken from [17]

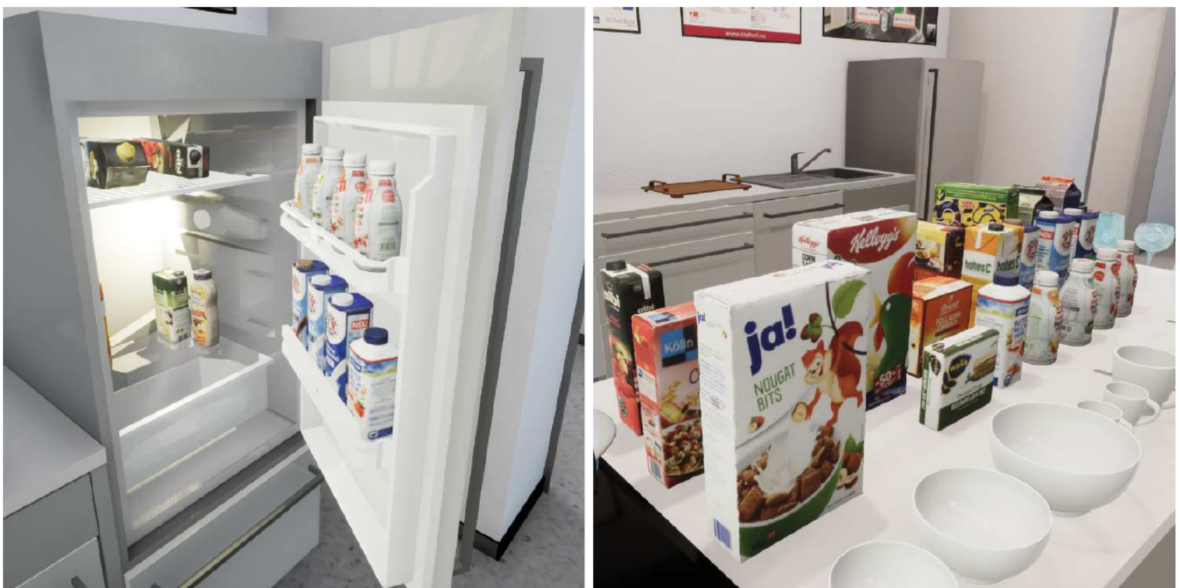


Figure 8: Stored objects in the fridge (left) and all usable objects on the kitchen island table (right). Figure taken from [11]

Since the motivation behind this bachelor thesis is to learn from specific persons, I started recording the VR data myself. Because of the time limitations of a bachelor thesis, the collected data set does contain only VR data from me. Since this bachelor thesis only collected data for the context breakfast, objects were used which are com-

monly on my breakfast table. The most used objects were the big bowl, big spoon, plate, cup, knife and cereal box, due to the fact that I eat mostly cereals for breakfast.

episode name	quantity	description
human-muesli-i[15]	5	cereals setup and ignoring pose of the human while placing
rob-muesli-i	11	cereals setup w. r. t. the robot base size by choosing safe standing poses
right_side_table_muesli_i	9	cereals setup only in the right corner of the table
full_breakfast_setup_i	11	breakfast setup with different objects
number of all episodes	36	

Table 1: Collected and exported episodes

Table 1 shows how many different episodes were collected[16]. Moreover, every row explains roughly what actions were recorded in VR. The "cereals setups" were executed in 25 VR experiments and used only the objects: big bowl, big spoon, cereal box, orange juice, milk, glass and cup. Since during the recording phase it was not clear which model would be the best fulfilling the required task, VR experiments were executed differently. In the eleven rob-muesli VR experiments I tried to move in the range and position capabilities of the robot. This means e. g. that for picking and placing of objects the human base should be further away from the drawer or table since the robot base is larger. The main reason behind this was to assure a "safe" dataset containing human poses which would be easily applicable on the real robot, if the model needed the robot positions. Every other episode was recorded without this restriction. Episodes in "human-muesli" and "rob-muesli " contain only breakfast settings for one person. The used objects in these episodes were mostly placed on the side of kitchen island showing towards the kitchen sink area.

Since another target in this bachelor theses included the model being powerful enough to represent the favorite object positions of one specific person, the episodes called "right_side_table_muesli" were recorded. These episodes were again only "cereal setups", but the used objects were only placed in the right bottom corner of the table as shown in Figure 9. This was done, because it was my favorite seating position.

Although only I was recording the VR data, I did setup the breakfast table in the episodes of "full_breakfast_setup" as shown in Figure 9 for more persons too. This was done, because the robot should learn setups for more than one sitting position.

[15]the suffix "i" in each row represents a number from 1 to the value of the quantity of collected episodes

[16]Episodes: `https://seafile.zfn.uni-bremen.de/d/131ec90a98ed401c9535/`

Moreover, the episodes “full_breakfast_setup“ used more objects like plates, forks and knives.

Figure 9: One full_breakfast_setup VR experiment shows how the kitchen island was placed with big bowls, spoons, cups, a plate, knife, fork, milk, cereal box and juice.

To query through the collected data, the JSON-files of these 36 episodes were saved in the data base MongoDB inside the knowledge processing system KnowRob. The documentation in [14] and an import script in bash[17] made this step more comfortable.

4.2.2 Querying of KnowRob

In the paragraph contribution 1.4 was already mentioned, that additional Prolog queries (see Section 3.2) were written in the CRAM package cram_knowrob_vr. These allowed with the communication interface json-prolog (see Subsection 3.1.1) and ROS, to query through the collected VR data and to export the positions of different used objects in a structured CSV-file. This file is presented in Table 2 and explained in the following.

Every row in Table 2 represents one object instance of a given object type that was used in one experiment. So if an object instance called Cup_jg04 of object type cup was used in different episodes, different samples were exported in Table 2. Since the built system should be able to learn object placements for different context e. g. breakfast, dinner and lunch, although I only recorded data for breakfast setups, the data had to be categorized accordingly. The first three columns context, kitchen name and human name in the CSV-file are filled the same entries for all samples. The context is BREAKFAST, the kitchen name is KITCHEN and the human name is THOMAS. Moreover, the last column in the CSV-file is named table name and has for every sample

[17]MongoDB import script: `https://github.com/hawkina/useful_scripts`

the value rectangular_table. All these columns were added, since it is possible that this dataset will be extended containing different contexts or kitchen setups. Moreover, the dataset allows to add VR experiments of other humans or of other kitchens containing different tables. These columns strictly exist to define a hierarchical structure in the built system, which is explained in Subsection 4.3.2.

The rest of the data in Table 2 is structured in the two subcategories. The first subcategory shows where the used object instances were stored in the given kitchen and the second subcategory shows where these used object were placed. The location column in both subcategories saved on which kitchen link, i. e. specific piece of furniture, the objects were before or after they were delivered by the human. The exact position before or after the object instances were delivered, is recorded by saving the X and Y coordinates in the global map frame of the kitchen. The Z coordinate was omitted, since the kitchen objects were always placed on the flat kitchen island called IslandArea. Moreover, the orientation of the used objects were exported too. The orientations represent the Z rotation of the object instances in Euler angles. Lastly, the arm which picked and placed the used objects in VR was saved. The data set contains in total 391 samples. After removing invalid object placements and objects which have to less object placements, the data set contains 357 samples.

Table setup for context BREAKFAST, kitchen KITCHEN, human THOMAS and table rectangular_table									
	storage location and pose				destination location and pose				
object type	location	x	y	orient.	location	x	y	orient.	arm
SpoonSoup	SinkDrawerLeftTop_05qp	0.96921	0.83987	3.10899	IslandArea	-0.70125	1.18529	-0.02311	RIGHT
SpoonSoup	SinkDrawerLeftTop_05qp	1.19926	0.82122	-3.09478	IslandArea	-0.67434	1.16612	0.07456	RIGHT
SpoonSoup	SinkDrawerLeftTop_05qp	1.07832	0.83979	3.14079	IslandArea	-0.75685	1.20909	-0.01101	RIGHT
...	...	...	...	...	...	...	...	...	...
SpoonDessert	SinkDrawerLeftTop_05qp	0.90577	0.96394	2.98784	IslandArea	-1.23516	0.88701	2.3251	RIGHT
SpoonDessert	SinkDrawerLeftTop_05qp	0.90577	0.96394	2.98784	IslandArea	-1.21988	0.88074	2.92369	RIGHT
...	...	...	...	...	...	...	...	...	...
KnifeTable	SinkDrawerLeftTop_05qp	1.39911	1.18568	3.07806	IslandArea	-1.1815	0.85523	-3.04633	LEFT
KnifeTable	SinkDrawerLeftTop_05qp	0.97931	1.18604	2.90333	IslandArea	-0.67431	1.89031	-0.11243	RIGHT
...	...	...	...	...	...	...	...	...	...
BowlLarge	IslandDrawerBottomLeft_nhwy	-0.7516	1.06283	-0.50437	IslandArea	-0.75099	1.06232	-0.50348	RIGHT
BowlLarge	IslandDrawerBottomLeft_nhwy	-0.7516	1.06283	-0.50437	IslandArea	-0.70388	1.09247	-0.5129	RIGHT
...	...	...	...	...	...	...	...	...	...
PlateClassic28	IslandDrawerBottomMiddle_H0F7	-0.66267	1.80376	-0.09634	IslandArea	-0.89732	0.74295	0.43176	LEFT
PlateClassic28	IslandDrawerBottomMiddle_H0F7	-0.66267	1.80376	-0.09634	IslandArea	-0.92993	0.67549	0.08258	RIGHT
...	...	...	...	...	...	...	...	...	...
GlassTall	SinkDrawerLeftMiddle_jqU4	0.96216	0.92014	0.00669	IslandArea	-0.84748	1.84471	2.5041	RIGHT
GlassTall	SinkDrawerLeftMiddle_jqU4	1.14985	0.81046	-0.00165	IslandArea	-0.80293	1.93634	-2.48835	RIGHT
...	...	...	...	...	...	...	...	...	...
exported samples: 391 and filtered samples: 357									

Table 2: Some samples from the exported episodes

4.3 Object Placement Learning Model

4.3.1 Assumptions

Since the built model should not exceed the scope of a bachelor thesis, some assumptions were made in the first hand. Firstly, the object size is not included in the data and neither is explicitly declared in the built model. Therefore, it can happen that objects overlay while placing. Moreover, the saved coordinates are dependent on the specific kitchen setup with specific furniture poses, although they could be recalculated if e. g. the table moves in the VR kitchen or in the kitchen of the Bullet simulation. Lastly, the data recorded only applies to the rectangular table used in the VR kitchen, otherwise objects may fall of the table. Although CRAM checks this unstable table placements, it would still lead to uncommon breakfast settings.

4.3.2 Architecture

The ROS package costmap_learning was implemented in Python and solves the tasks introduced in the Hypothesis 1.2 by creating ROS services allowing the CRAM node to query for information saved in costmap_learning.

4.3.2.1 Component Level

Figure 10 shows the communication interfaces between the two ROS nodes costmap_learning and CRAM. In the CRAM package cram_pr2_pick_place_demo are the kitchen environment and the simulated robot PR2 loaded in the simulation Bullet.

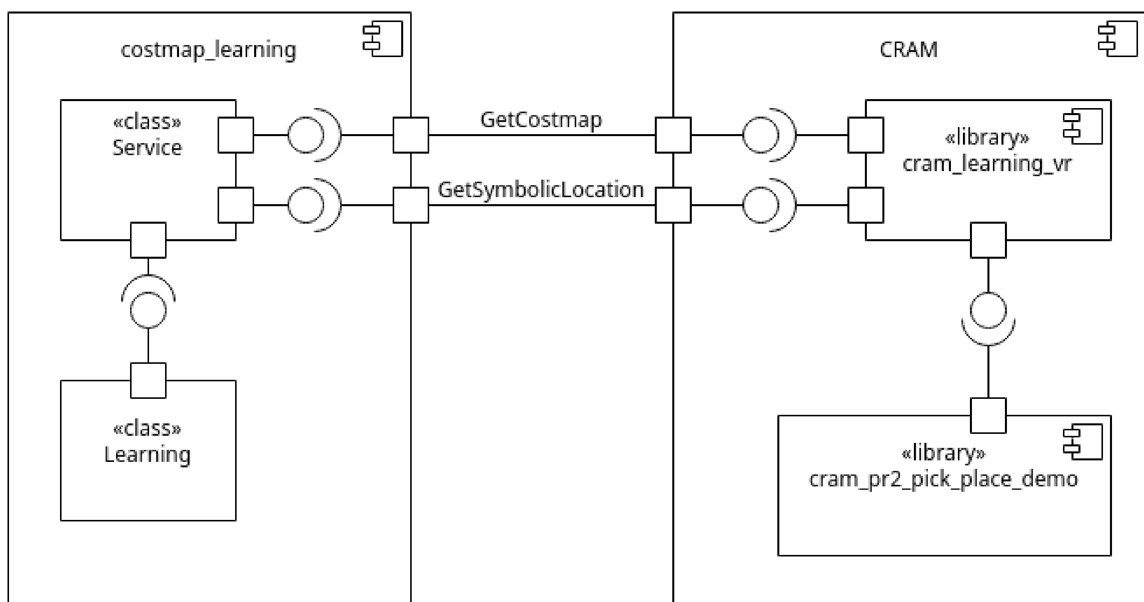


Figure 10: The component diagram showing the services GetCostmap and GetSymbolicLocation of the built system costmap_learning

Moreover, different objects, which where were used in the VR experiments too, can be used for pick and place tasks. CRAM has a general plan for executing pick and place tasks and currently uses heuristics to infer object placements during runtime. To replace those, I wrote the CRAM package cram_learning_vr, which queries with the services GetSymbolicLocation and GetCostmap the built ROS package costmap_learning as shown in Figure 10. Hence GetSymbolicLocation and GetCostmap are ROS services, specific information from the robots knowledge and the location designators need to be passed, so that costmap_learning can use it for accessing the correct object placements. The ROS services are explained in the following.

GetSymbolicLocation To get the symbolic storage or destination of an object, the service GetSymbolicLocation uses the following parameters:

- object type, e. g. BOWL
- kitchen, e. g. KITCHEN
- table, e. g. rectangular_table
- context, e. g. BREAKFAST
- human, e. g. THOMAS
- storage-p, e. g. False, since the destination is wanted

In costmap_learning is the symbolic destination of the object inferred with values from the location designator, which are passed in the above parameters. For inferring the symbolic storage placements only the object type and kitchen need to be given, since normally all humans using the same kitchen place the objects of the same type at one specific place. The service returns the symbolic location as a string to CRAM. In the CRAM package cram_learning_vr the returned string will be passed into a location designator to represent the location in CRAM.

GetCostmap To get the distribution of the used objects, the service GetCostmap uses the following parameters:

- object type
- kitchen
- table
- context
- human
- storage-p
- placed-object-types
- placed-object-coordinates

This service needs additionally, the object types and coordinates of the objects placed on the table IslandArea. The objects coordinates have to be in the same frame as the coordinates in Table 2. After validating the input parameters of this service, the Learning class calls functions, which access the learned models to get the needed placements as a distribution. The subsymbolic destination or storage placements are then returned to CRAM. In the CRAM package cram_learning_vr the returned subsymbolic placement information of a given object type will be passed into a Location-Costmap object (see Subsection 3.4.4), thus CRAM can sample a pose for the object of the given object type. In Chapter 5 Location-Costmap objects of different from costmap_learning returned distributions are visualized.

Although the GetCostmap message includes exactly the same information as the GetSymbolicLocation message, these services are not combined. This has two reasons. Firstly, the services are modular and it is clear which type of information they return, instead returning the symbolic and subsymbolic information in one message. Secondly, CRAM does need to know for the planning of transporting tasks the objects symbolic storage and destination location. After the transporting plan could be evaluated, the following delivering action requires the subsymbolic destination placements for choosing a robot place from which the placing task of the object could be executed successfully.

4.3.2.2 Class Level

The package costmap_learning has six Python classes shown in Figure 11. The architecture of this system is modeled as a hierarchical structure starting with the class Kitchen. The class Kitchen is represented by an unique name and embodies all humans, which performed VR experiments. The Human objects each have for every table they set a Settings object. These Settings objects represent different table settings e. g. breakfast, dinner or lunch. For each of these settings different VRItem objects are assigned. Therefore, costmap_learning saves different object placements for different kitchens, humans and contexts. One VRItem represents one object type used in the VR experiments e. g. the SpoonSoup. Therefore, in one VRItem object all placements are saved for the given specific object type, kitchen, human, table and context (e. g. breakfast). This means, that there are more representations e. g. of the SpoonSoup object, if e. g. two humans set the table or if the context was changed.

The concrete placement information in one VRItem object is splitted in two Costmap objects: one representing the storage placements and the other the destination placements. Moreover, next to the concrete placements the symbolic storage and destination locations of the VRItems are saved too. The symbolic destination location is always IslandArea, since every used object in VR was placed on the same table in the kitchen. Therefore, the robot does not only know on which surface the object was placed or stored, but has a distribution of the placements of the used objects too. Each concrete storage and destination placement of the given object type is represented by a

Costmap object. The Costmap object copies the parameters of the VRItem object and models the concrete storage or destination placements with a GMM. Furthermore, the Costmap objects represent the orientations of the placed and stored objects too.

Lastly, the class OutputMatrix was implemented to export the Costmap objects in a matrix, which could be sent back via ROS and be visualized immediately in the simulation of Bullet inside of CRAM.

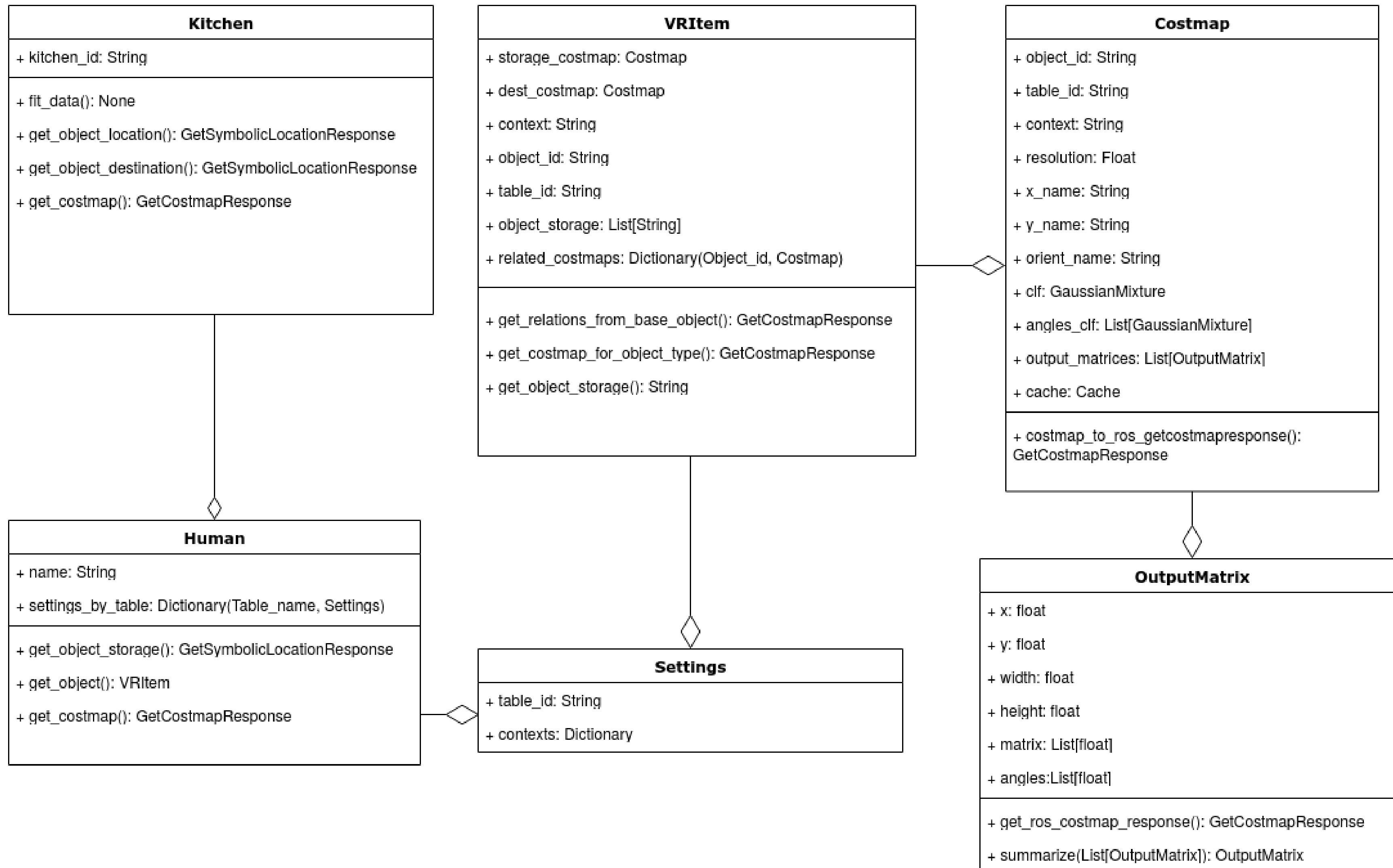


Figure 11: The class diagram of the built system costmap_learning

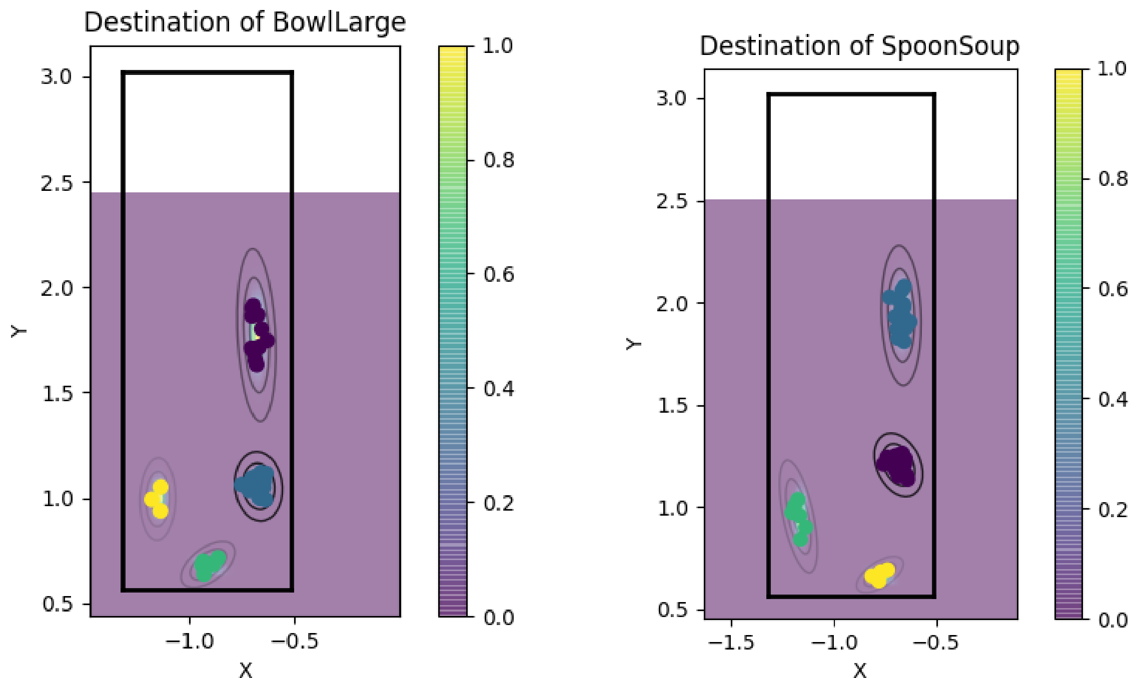
4.3.3 Implementation

In Subsection 4.3.2 were already the three most important attributes of an VRItem object explained: the concrete storage placements in storage_costmap, the concrete destination placements in dest_costmap and the symbolic storage labels in object_storage. The symbolic storage labels are saved in a sorted list and are therefore easy to access and save after getting the needed values for the parameters: kitchen name, human name, context and object type. The concrete placements of a VRItem object are in storage_costmap and dest_costmap each represented by a Costmap object.

4.3.3.1 Model

Each storage and destination Costmap object needs to contain a model representing the coordinates of the object placements. This model should be exportable in such way, that it could be easily converted into Location-Costmap objects in CRAM. This would ensure, that the planning framework will not miss any information represented in the chosen model.

The first step in learning the object coordinates is to cluster the coordinate points shown as different colored points in the Figure 12.



(a) The destination Costmap of the VRItem BowlLarge visualized

(b) The destination Costmap of the VRItem SpoonSoup visualized

Figure 12: Visualized Costmap objects of different VRItem objects BowlLarge and SpoonSoup

For this different clusterings methods like KMeans, SVM or the Expectation-Maximization (EM) algorithm were considered. In the Costmap class the EM algorithm with the initialization points of KMeans is used. The SVM was not chosen because the shape of

the clusters is not complex, but has either the shape of a circle or ellipse, which can be observed in the Figure 12. Moreover, it could not be used easily with the Location-Costmap objects from CRAM since the SVM of a object would not represent a distribution of the object placements, but only the border of the object placements. The EM algorithm provides after converging mean and covariances values, which can be used to create a distribution of the object coordinates. Since the shape of the clusters are not complex, two generative models were considered: Naive Bayes (NB) and GMMs.

Both classifiers are probabilistic and could export a discrete distribution for CRAM. The NB classifiers would be used by creating a fitted NB classifier for every cluster of the object placements. Therefore, in Figure 12(a) the BowlLarges destination placements would be represented by four NB classifier, each representing the concrete placements of the different colored points. With a GMM, these different colored points would be modeled by one GMM object with four components. Thus, the integration in the Costmap class would either need n NB classifiers or one GMM classifier with n components for modeling n clusters. Both classifiers would use the coordinates as features and could return for every point in the distribution a probability. Furthermore, both classifiers allow sampling.

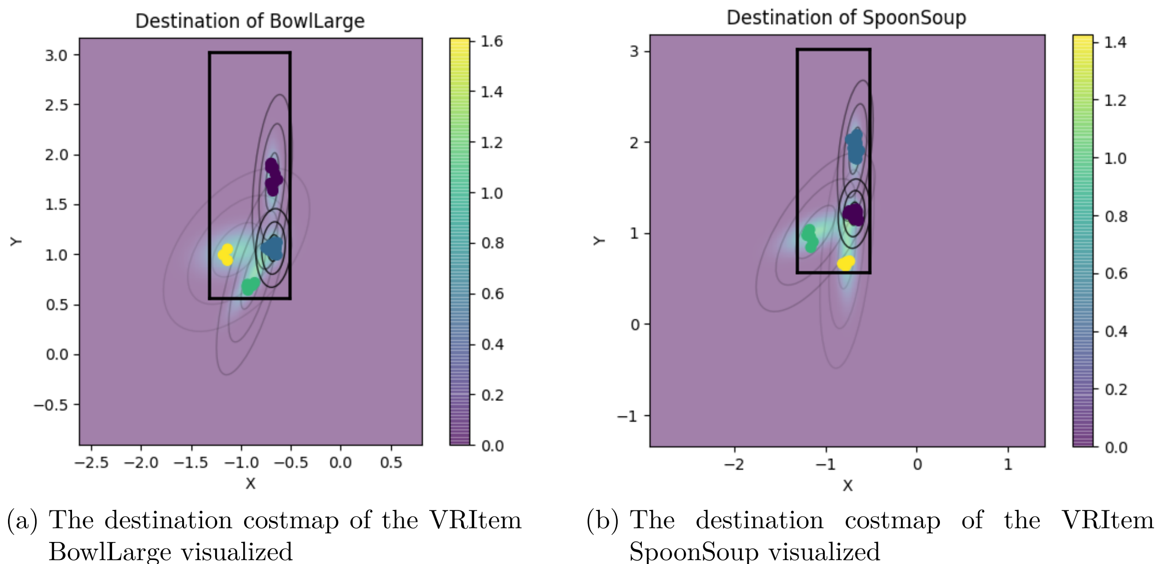
In the Costmap class GMMs are used to model the discrete object placements. The main reason is, that the discrete distribution for the placements of one object type represented with different NBs classifiers would not represent the collected object placements. Firstly, because two clusters would influence each others exported probability distribution, if these are connected through an edge which is perpendicular to the X or Y axis. This can be explained by the independency between features in the NB classifiers. E. g. let us assume, that the object placement coordinates in Figure 12(a) are modeled with NB classifiers. Then, the distribution matrix of the NB classifier with the blue points would be influenced by the NB classifiers of the yellow and purple points. This can be explained due to the fact, that the X probabilities of the NB classifier with the purple points, match with the X probabilities of the blue points. Furthermore, the Y probabilities of the NB classifier with the yellow points, match with Y probabilities of the blue points. This would indicate a dependency from the NB with the purple and yellow points, towards the NB classifier with the blue points, although this dependency does not exists. Moreover, this would lead to slightly changed probability distributions if one or more of these NB classifiers would be masked by an already placed object.

Secondly, it could not be differentiated which seating or placing position represented by at least one NB classifiers is preferred.

GMMs do not have these two problems. GMMs are represented by a finite and fixed number of components. Each component is represented by a Gaussian distribution. The Gaussian distributions in one GMM each depend on a mean and variance value. The fitted GMMs used for modeling object placements have only the X and Y coordinate as features. Therefore, each mean is represented by coordinate point and each

variance holds a two times two covariance matrix. Since the different covariances allow to represent the expectation of dependent random variables, the components of each GMM do not influence each other.

Moreover, the seating positions are with GMMs not all the same. Since the components of one GMM are weighted, these value can be used to determine and encode the favorite seating position. Furthermore, GMMs are clustered with the EM algorithm after initializing the means with KMeans. In costmap_learning the GaussianMixture class from the python library scikit-learn [20] is used. Experiments with the BayesGaussianMixture model from scikit-learn shown in Figure 13 concluded, that the BayesGaussianMixture class is not suitable for the collected object placements, since the covariance values of the clustered points were too large.



(a) The destination costmap of the VRItem BowlLarge visualized

(b) The destination costmap of the VRItem SpoonSoup visualized

Figure 13: Visualized destination costmap objects of the different VRItem objects BowlLarge and SpoonSoup. The used model in the destination costmap was the BayesGaussianMixture from sklearn [20], which created covariances not fitting to the clusters of the points.

4.3.3.2 Costmaps

One important task for the Costmap objects is to save the storage or destination placement of a VRItem object by representing the coordinates and orientations. Every VRItem represents given the kichen name, human name, context and table, the placed objects of one object type. The X and Y coordinates of one object type are split in destination and storage placements (see Table 2) and therefore are each modeled by one Costmap object. The Costmap objects in costmap_learning use as explained in Subsection 4.3.3.1 GMMs to model the concrete object placements. The GMM uses only the X and Y coordinates of the given object as features. The number of components for the GMM is calculated dynamically with the silhouette score.

Figure 12 shows the destination placements of the two VRItem objects BowlLarge and SpoonSoup. The black outline represents the borders of the table IslandArea. Both GMMs of the BowlLarges and SpoonSoups destination placement have four components. Each component clusters the object coordinates visualized in different colored points. Every component has its own multivariate Gaussian distribution, which is visualized with three ellipses around the mean point of the component. The ellipses around the mean point represent different deviations from the mean point. The first and smallest ellipse represents the deviation of $\pm 1\sqrt{\sigma}$, the second and larger ellipse of $\pm 2\sqrt{\sigma}$ and the third and largest ellipse of $\pm 3\sqrt{\sigma}$ from the mean point. Since costmap_learning does not return the parameters of the Gaussian distribution, but the distribution outputed in a matrix, this matrix is visualized too as shown in the Figures 12. The bar on the right in each of the Figures 12, represents the normalized values of the learned GMMs density distribution. Therefore, the background colors from purple to yellow show the GMMs different values for different coordinates points. Since the matrices in the Figures 12 have limited sizes, the white background shows the coordinates which are not covered in the matrices.

Since a human did setup the breakfast table, one cluster in the BowlLarges and SpoonSoups destination placements, represents one seating position on the table. The cluster at the short side at the bottom and at left side of the table show only one seating postion. At the right side of the table the destination placements of both objects show each two cluster and therefore two seating positions. All clusters of points presented in Figure 12 are well covered by different Gaussian distribution. Since the third and largest ellipse of each component reflects appropriately that 99.73%[18] of the values lie within the component, the GMMs model the placement of the Spoon and Bowl well as shown in the Figures 12. Moreover, the heatmap from purple to yellow represents the exported density of the GMM models accordingly.

The destination placements of the BowlLarge and SpoonSoup are overlayed in Figure 14. In this Figure the seating positions are still clearly separable from each other. Moreover, the overlayed destination placements show, that the SpoonSoup was mostly placed on the right side of the BowlLarge. Although the clusters of the different object types are still separable, a certain overlay exists at each seating position between the two different clusters.

[18]The empirical rule describes that 99.73% of the values lie within the margin of $\pm 3\sqrt{\sigma}$ from the mean

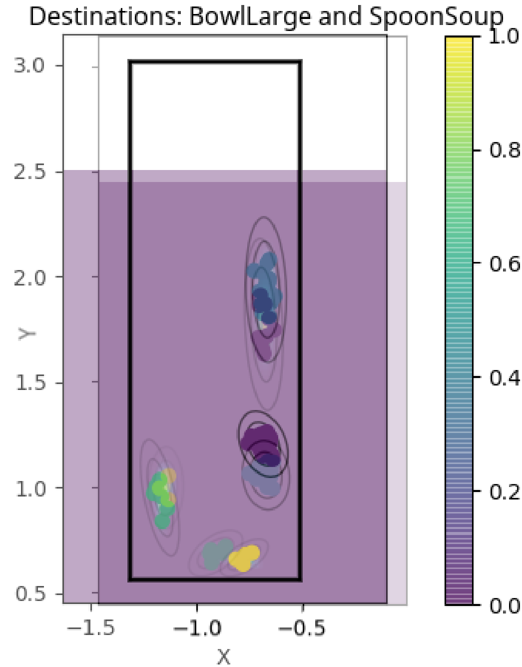


Figure 14: The destination costmaps of BowlLarge and SpoonSoup overlayed

Next to the positions of a given VRItem object, the orientations are saved in the destination and storage costmaps too. Therefore, the objects storage and destination orientations are like the placements saved in different Costmap objects.

The orientations were represented as Euler angles. Therefore, every orientation was described by the rotation around X, Y and Z axis of the object. Since the used objects in VR were only placed on flat surfaces, the rotation around the X and Y axis were ignored. To choose a model for orientations of a costmap object, the assumption was made, that the rotation around the Z axis is normally distributed. Due to this characteristic, each Costmap object has a list of GMMs each representing the orientation of one component in the position GMM of the Costmap object. Since the destination placements of the BowlLarge objects are bundled in four clusters as shown in Figure 12(a), the placement GMM has four components. Each component in the placement GMM has another GMM representing the orientation of the bundled points, thus it can be possible that multiple preferred orientations exist for a specific object.

Since my implementation allowed to work more easily with GMMs, these were used for the representation of orientations too. Every orientation GMM has only one component and the orientation is the only feature. In Figure 15 are the four different orientations of a SpoonSoup each representing orientations for a specific seating at the table on the corresponding side of the table shown. The GMMs at the top in Figure 15 are connected to the SpoonSoups placement components which are on the right side of the table as shown in Figure 12(b). The reason for that is, that spoons which were placed in the area of the purple and blue points, were always perpendicular to the next close table border placed in such way that the top of the spoon showed away

from the next close table border. The orientation values for these placements should be, if the spoons were always perfectly placed, zero degrees.[19] Therefore, the spoons represented by the yellow points and green points in Figure 12(b) should be rotated around $\frac{\pi}{2}$ and π. Thus, the orientation GMM at the bottom right in Figure 15 represents the orientation of the yellow points and the orientation GMM at the bottom left represents the orientation of the green points. Due to the small variances of the orientation GMMs and the different means, this model shows that the orientation of the SpoonSoup is important for breakfast table settings. Similar can be observed in Figure 29 representing the orientations of the object KnifeTable.

The other objects e. g. the bowl, milk and cup shown in Figures 30, 31 and 32 do not tend to have a preferred orientation for the placements since the variances are not small and the means are not different. The plate object e. g. has as shown in Figure 16 always nearly the same mean and mostly high variance values. The orientation GMMs with smaller variances compared to the orientation GMMs of the same object, could therefore indicate that the orientation matters for some placing positions or that for these particular placing positions the pose of the human or his hand while picking or placing did not rotate. Since the orientation GMMs with smaller variances correspond to the placements near the plates storage poses, the latter could explain the small variance values.

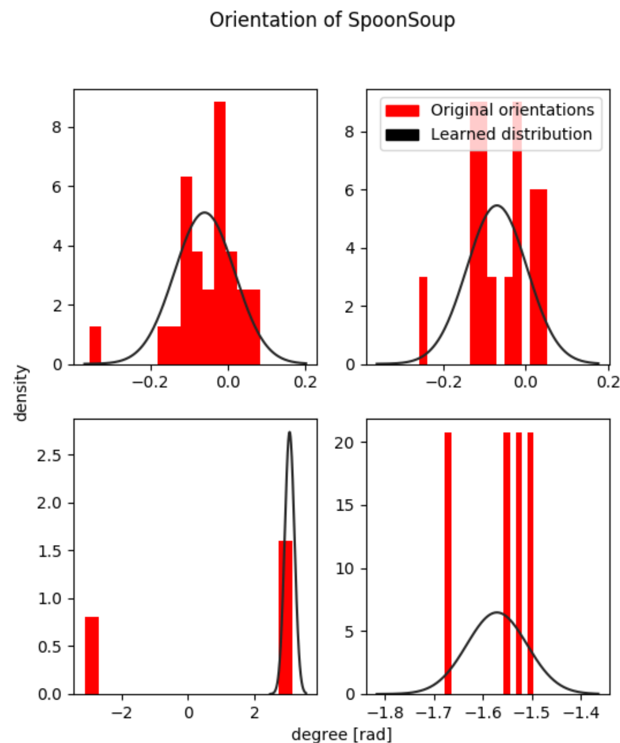


Figure 15: The destination orientation distributions of the object SpoonSoup

[19]the angles in the Bullet simulation in Chapter 5 have an offset of π since the kitchen in the Bullet simulation is compared to the kitchen in the Unreal Engine rotated around the Z axis

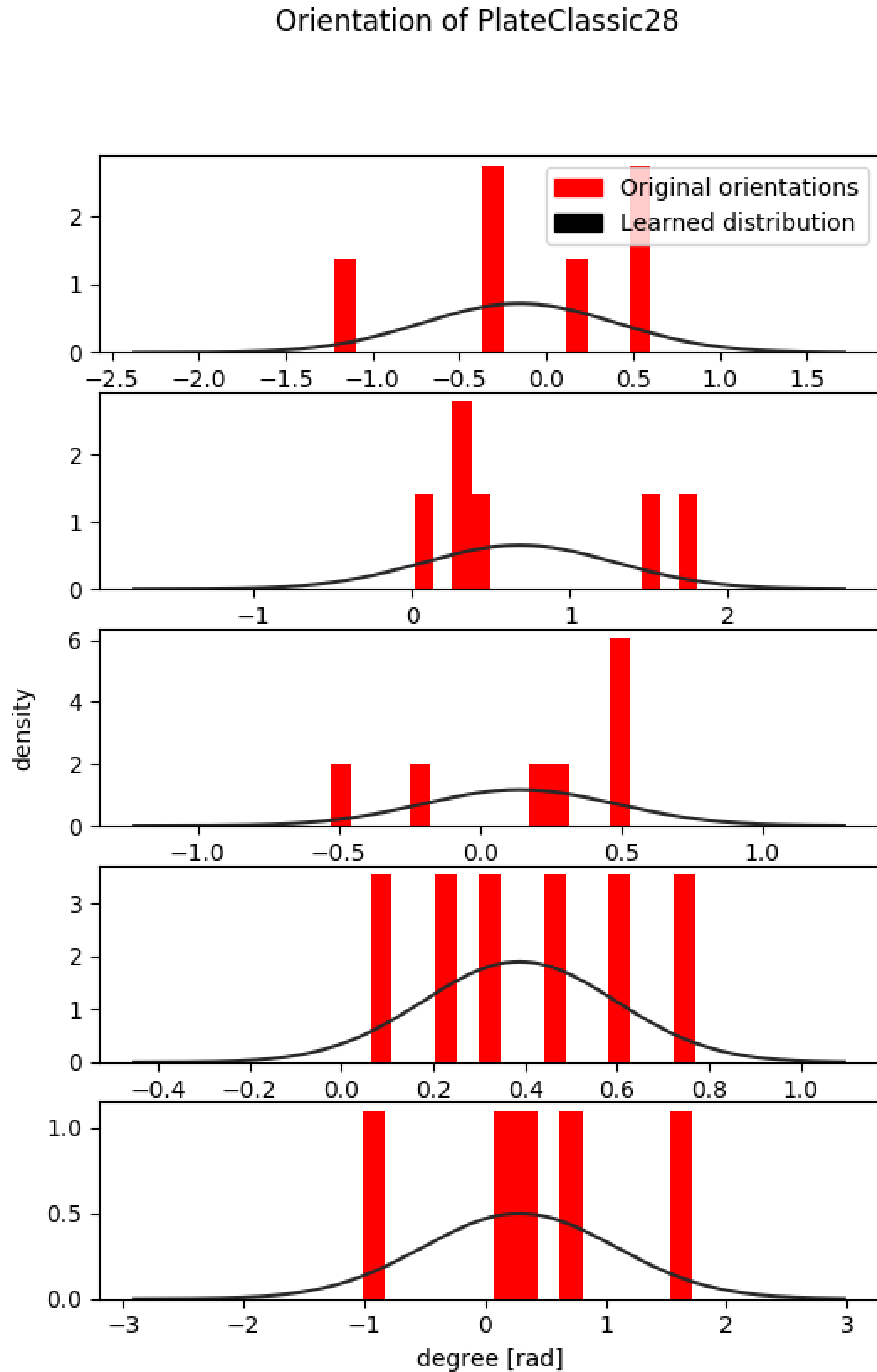


Figure 16: The destination orientation distributions of the object PlateClassic28

4.3.3.3 Related Costmaps

As presented in the Hypothesis 1.2, the ROS package costmap_learning is able to respond with relational costmaps, if objects are already placed on the kitchen island. E. g. if one bowl was placed on the table in the simulation Bullet inside of CRAM and the robot was ordered to place a spoon, the distribution for the spoon should be next to the bowl that is already on the table, since the human did it in the VR experiments too. For this a relational costmap is calculated, which will be returned, so the spoon gets placed like in the VR experiment.

Before related costmaps can be calculated, first one must understand what in particular is in relation with each other. Every costmap object contains a placement GMM with n components. These n components, which each represent the clustered placement coordinates, are connected to one of the m components of another object type. In case of bowls and spoons n and m are the same, but, e. g. the cereal box object has only one component and is placed on the table independent of how many bowls are used. To choose for each of n components one component of the other object type, the euclidean distance between the means of these components are calculated. The closest component of the other object type is then connected to the compared component. This is done m times for each of the n components. These connected components of two different object types are then called relational costmaps. In costmap_learning relational costmaps between every used object type are calculated and saved in the VRItem object. Therefore, every VRItem contains a list with related costmaps. This procedure would create eight relational costmaps between the destination costmaps of the BowlLarge and SpoonSoup. Four relational costmaps would be saved in the VRItem object of BowlLarge and the other four would be saved in the VRItem object of SpoonSoup. E. g., the component with purple points of the BowlLarge would be connected to the component with blue points of the SpoonSoup, and vice versa. The connection from the BowlLarge would be saved in the VRItem object of BowlLarge and the connections from the SpoonSoup in the VRItem object of SpoonSoup.[20] In costmap_learning two different approaches were implemented to calculate the relational costmaps between different object types.

First Approach In this approach the relational costmaps are only represented as symbols. E. g. the relational costmap `BowlLarge0<->SpoonSoup1` denotes that the first component of BowlLarges destination placements and the second component of SpoonSoup destination placements form a relational costmap. Thus, if a object of type BowlLarge was placed in the first component of the BowlLarge, the second component

[20]BowlLarges related costmaps towards SpoonSoup:
$[Purple \rightarrow Blue, Blue \rightarrow Purple, Green \rightarrow Yellow, Yellow \rightarrow Green]$
SpoonSoups related costmaps towards BowlLarge:
$[Purple \leftarrow Blue, Blue \leftarrow Purple, Green \leftarrow Yellow, Yellow \leftarrow Green]$

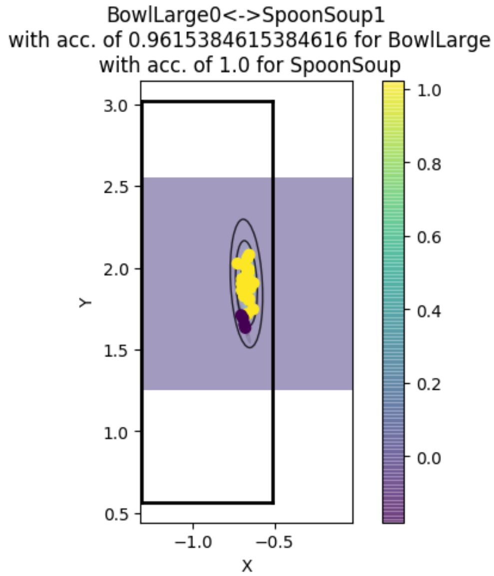
of the SpoonSoup would be returned, if the robot was ordered to place a SpoonSoup object. If two BowlLarge objects were already placed and their placements both were in two different BowlLarge components, two SpoonSoup components from two different relational costmaps would be returned allowing the robot to decide next to which BowlLarge the SpoonSoup should be placed. If one BowlLarge and SpoonSoup were already placed next to each other and satisfy the relational costmap saved in costmap_learning, a cut destination costmap[21] of the requested SpoonSoup or BowlLarge would be returned. The four relational costmaps between the BowlLarge and SpoonSoup are visualized in Figure 14. Since this approach has less assumptions as the second approach and is more appropriate for representing relations between the different costmaps, it is used in costmap_learning. All possible use cases are covered in the Chapter 5.

In the following a more complex approach that contained more information is explained. Since it is still possible, that the components of relational costmaps overlay greatly, another approach has been developed and implemented, which was very promising but had a number of issues. Although it was later discarded and is not used, I would still like to explain it because it shows the GMMs useful capabilities and limits.

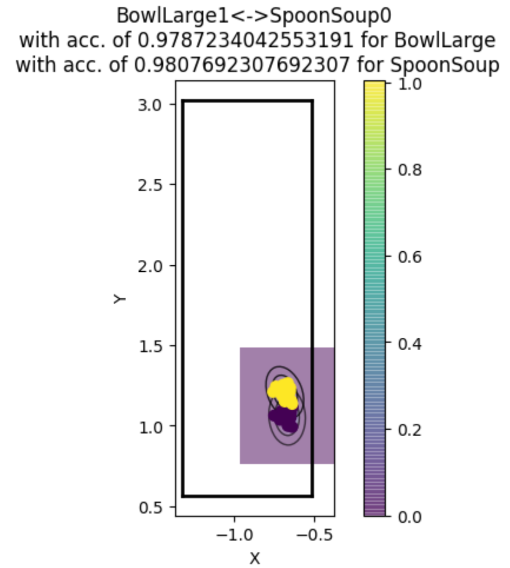
Second Approach This approach saves additionally to the symbolic representation of the relation costmap a new GMM. Instead of using the components of the destination placements, the related components get clustered again in a GMM with two components. This means that the relational costmap `BowlLarge0<->SpoonSoup1` would create a new GMM with the first component of the BowlLarges destination placements and the second component of the SpoonSoups destination placements. Although, the object placements are labeled, the new relation costmap GMM would cluster the coordinates in the both components again in two components. This behavior was wanted, hence it could reduce the overlapping between two components in the relational costmap. To get the relevant component from the relational component, costmap_learning would calculate which component represents the placement of the BowlLarge the most likely. Let us assume, that e. g. a BowlLarge was already placed on the table IslandArea in the kitchen and the robot is now ordered to place a SpoonSoup too. Figure 17 shows four plotted relational costmap objects representing the relation between the BowlLarge and SpoonSoup. During runtime costmap_learning would on the basis of the BowlLarges pose on the table, calculate which of the eight components in Figure 17 represent the placement of placed BowlLarge the best. If the component with the highest probability was chosen, costmap_learning returns the other component in the relational costmap. In case that e. g. the component with purple points from `BowlLarge1<->SpoonSoup0` as represented in Figure 17(b) represented the Bowl-

[21]Cut costmaps contain only one to $n-1$ of n components.

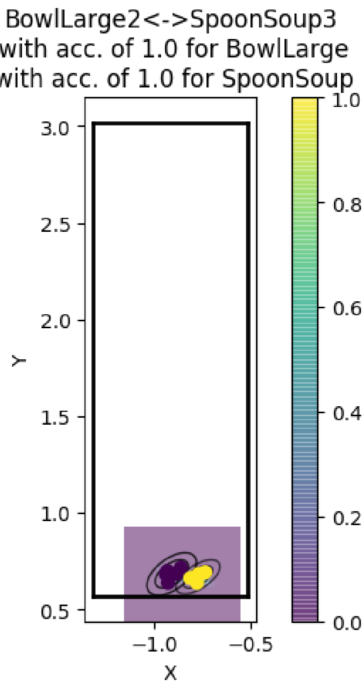
Larges placement the most, the component with the yellow points would be returned for the placement of the SpoonSoup.



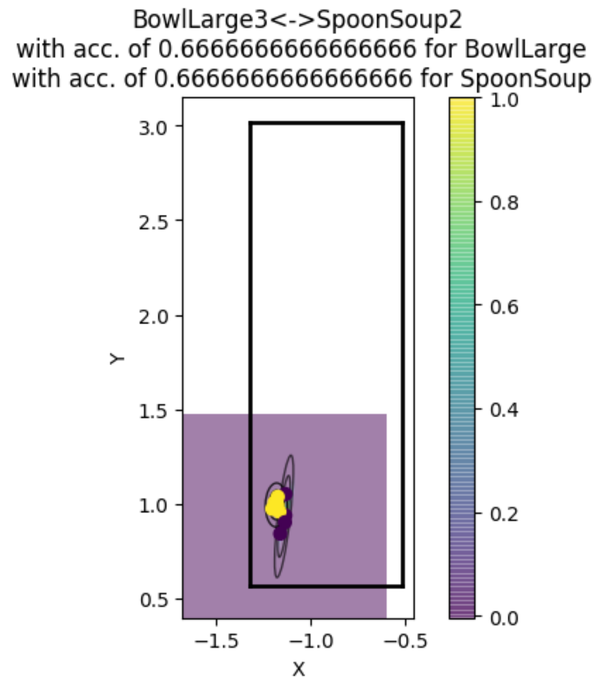
(a) This relation costmap consists of the component 0 of BowlLarge and the component 1 of SpoonSoup



(b) This relation costmap consists of the component 1 of BowlLarge and the component 0 of SpoonSoup



(c) This relation costmap consists of the component 2 of BowlLarge and the component 3 of SpoonSoup



(d) This relation costmap consists of the component 3 of BowlLarge and the component 2 of SpoonSoup

Figure 17: Visualized relational costmap objects between the different VRItem objects BowlLarge and SpoonSoup

The issues with this approach are presented with the relational costmaps shown in Figure 17. Because the BowlLarge destination costmap has four components, four relational costmaps between the BowlLarge and SpoonSoup were created.[22] Since the relational costmap clustered again the points in the components of the SpoonSoups and BowlLarges destination costmap, the accuracy for each relational costmap was calculated. Although, not all placements of the BowlLarge and SpoonSoup are represented correctly in the relational costmap of `BowlLarge1<->SpoonSoup0` shown in Figure 17(b), the two components in the relational costmap seem to distinguish better the relation between SpoonSoup and BowlLarge, since the overlapping could be reduced. The difference of overlapping between the relational costmap `BowlLarge1<->SpoonSoup0` and the components of the destination costmap of BowlLarge and SpoonSoup was not compared. Due to the major problems of this approach with the related costmaps, the difference in overlapping was neglected. This approach seemed only to work with clusters, which were already good enough to be distinguished from each other as represented by the relation costmap of `BowlLarge2<->SpoonSoup3` in Figure 17(c). Since the clustering ignored the labeled points, it is possible that components which overlay greatly, are clustered together. An example of this behavior is presented by the relational costmap `BowlLarge0<->SpoonSoup1` in Figure 17(a). Moreover, it would create unusable SpoonSoup and BowlLarge placements. If e. g. the BowlLarge was placed in the component with the yellow points, costmap_learning would always return the small costmap with the purple points as placement distribution for the SpoonSoup. This would be the same, if the placed object was the SpoonSoup first, since all SpoonSoup placements are in the component with the yellow points.

Lastly, the new created relational costmaps could create components, that are hard to understand. The `BowlLarge3<->SpoonSoup2` relational costmap shown in Figure 17(d), clustered the placements of BowlLarge and SpoonSoup in two components. Either one component has $\frac{2}{3}$ of the BowlLarge placements and the other has $\frac{2}{3}$ of the SpoonSoup placements or one component has each $\frac{2}{3}$ of the SpoonSoups and BowlLarges placements.

4.3.3.4 Algorithm

Since the objects destination placements depend on the human, the calculation returning the destination placements is mostly implemented in the Human and VRItem class. The algorithm for choosing the components which should be returned is explained roughly in the following.

The object type for which a costmap should returned to is called in the following o. First the algorithm checks, if every placed object has the object type o. If this is

[22] The other four relational costmaps between SpoonSoup and BowlLarge are here ignored.

true, the destination costmap of the object type o is returned. Components, which are already covered by the placed objects, are cut out.

Otherwise, the algorithm starts by choosing a placed object type, which is called o_p in the following. The algorithm tries then to find any placed object with the object type o_p, which might have a free relational component for objects with the object type o. Since the relational component between o and o_p could be already masked by another placed object of type o, o_p will be changed to another placed object type if no free relational costmap was found. If free relational costmaps were found, the components for the object type o from the relational costmaps are returned.

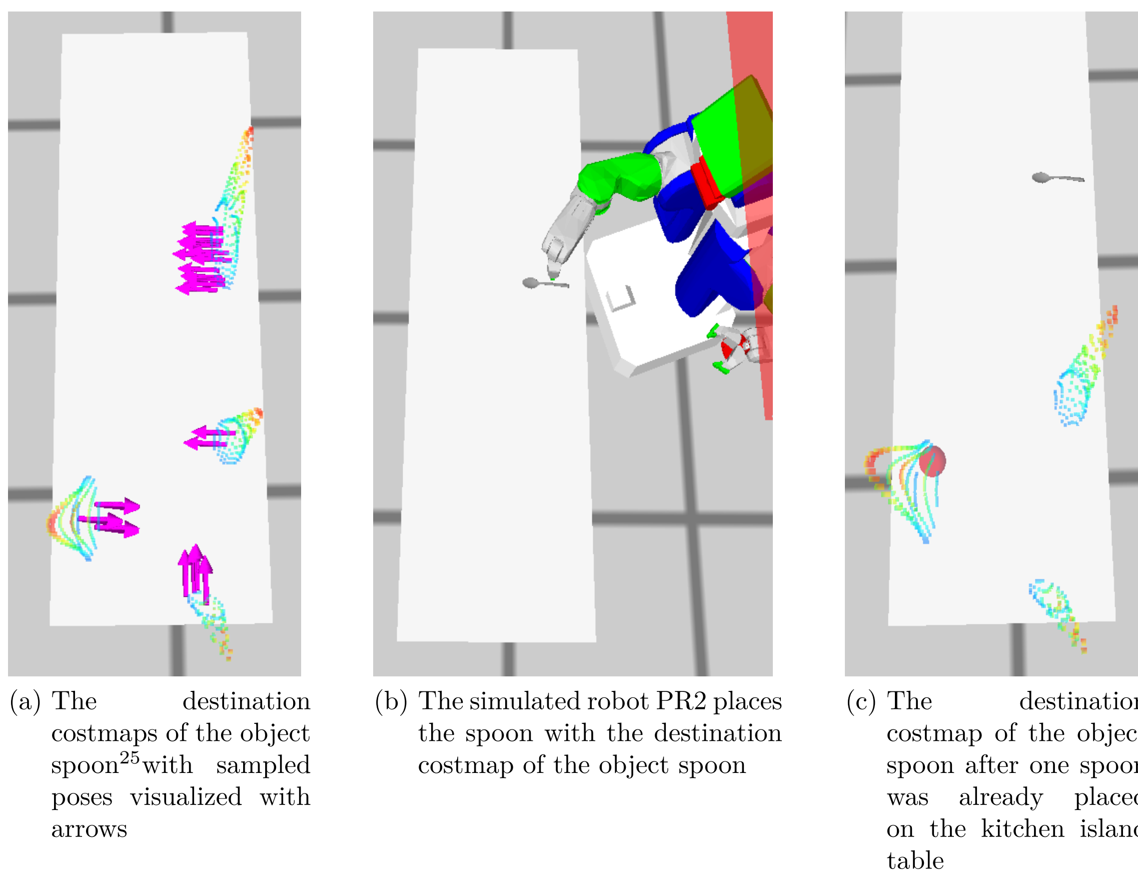
After trying unsuccessfully every placed object type, a cut destination costmap of the object type o is returned.

Since the visualized destination costmaps in CRAM are used to evaluate the built system costmap_learning, the algorithm returning these destination costmaps is explained with examples in Chapter 5.

5 Evaluation

The proposed and implemented pipeline was evaluated by requesting object placements in different kitchen states, and checking if the responded distributions were suitable for breakfast settings. For this different placements for the spoon and bowl are presented in the following.

The Figure 18(a) shows the destination costmap[23] of the spoon after the initialization of the kitchen as a heatmap in the simulation Bullet. Since no object is placed on the kitchen island table, all components of the GMM in the destination costmap of the spoon were returned. All four components are in CRAM strictly distinguishable and represent clearly different seating positions. The simulated robot PR2 will sample one pose from the visualized components and place the spoon accordingly as shown in Figure 18(b). Since each of the four components has its own orientation representation, the different components orientations can be visualized. The purple arrows in Figure 18(a) show the position and orientation of the sampled spoon poses[24]. All arrows point strictly away from the next close table border.



(a) The destination costmaps of the object spoon[25]with sampled poses visualized with arrows

(b) The simulated robot PR2 places the spoon with the destination costmap of the object spoon

(c) The destination costmap of the object spoon after one spoon was already placed on the kitchen island table

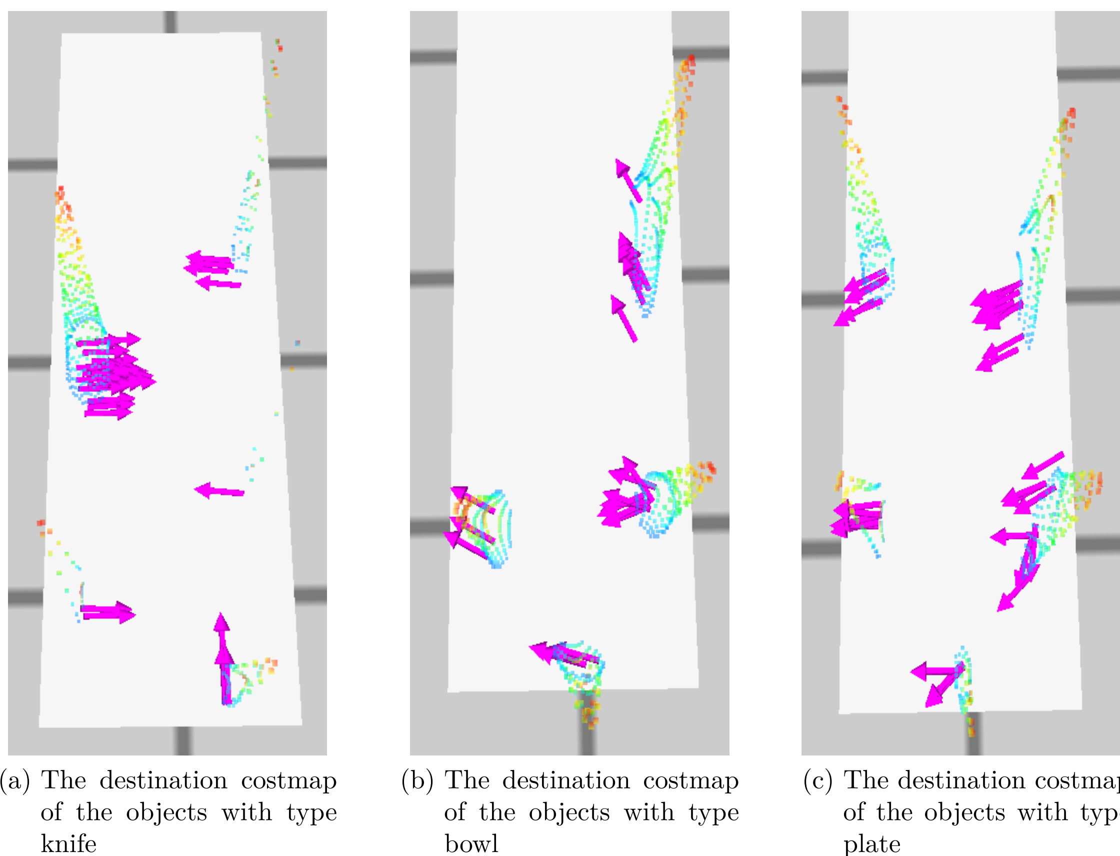
Figure 18: Visualized costmaps of the object type spoon

[23]the returned distribution from costmap_learning was represented in CRAM by a Location-Costmap object 3.4.4. Due to readability, henceforth Location-Costmap objects are referred to as costmaps.

[24]the Z rotation modeled in the components orientation GMM was sampled with the box-muller transform

Moreover, arrows in the same component are nearly parallel to each other, although the orientation differentiate greatly in general[26]. Therefore, the spoons orientations fit to the representation as shown in Figure 15 and as explained in Subsection 4.3.3.2.

Figure 19(a) visualizes the destination costmap and sampled poses for the object knife. Once more, the orientations of the purple arrows show that the orientation of the knife is dependent from the component and barely changes in the component too. This was expected too, since the orientation Figure 29 from costmap_learning indicated the orientation of the visualized arrows. Lastly, the Figures 19(b) and 19(c) show visualized samples from the destination costmap of the bowl and plate. The orientation of the arrows does not change drastically inside the different components as shown in the bowl and plate components in the Figures 19(b) and 19(c). Nevertheless, this does not indicate, that the orientation matters for the bowl and plate, since the means of the components orientations do not change drastically for both objects compared to the means of e. g. the SpoonSoups components orientations as shown in Figure 15.



(a) The destination costmap of the objects with type knife

(b) The destination costmap of the objects with type bowl

(c) The destination costmap of the objects with type plate

Figure 19: More destination costmaps for objects with the types knife, bowl and plate

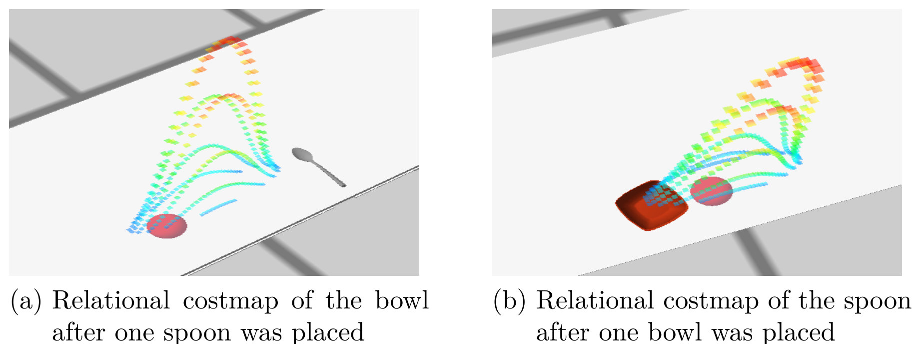
Since costmap_learning allows to send additional information about the changed environment, the robot can access cut destination costmaps leading to more efficient

[25]The objects of type spoon, bowl, knife and plate in the Bullet simulation are equivalent to the objects SpoonSoup, BowlLarge, KnifeTable and PlateClassic28 from the Unreal Engine

[26]orientations differentiate greatly, if the difference between these are not anymore in the margin of $\pm\pi/2$

planning. If the robot wants to place a spoon object after placing already one at the kitchen island table, costmap_learning will not respond with four spoon components, but with three as shown in Figure 18(c). Since the robot knows, that it placed one spoon object on the kitchen island table, it passes this information to the ROS package costmap_learning. costmap_learning checks in which of the spoon components the spoon is most likely placed in and cut it out of the returned distribution. Therefore, three components get returned for the placing of the spoon. The Figures 33(a) and 33(b) represent the same behavior, but with a bowl instead. Since the ROS package costmap_learning does not save the state of the kitchen, but uses only the inputed placement information, it can theoretically be used with more robots using the same kitchen too.

Let us assume, that after the first spoon was placed, a bowl should be placed on the kitchen island table. Since the pose of the placed spoon gets again transmitted to costmap_learning, the built package can check now which relational costmaps represents the wanted placement the best. First costmap_learning finds the component of the spoon which covers most likely the spoons placements. After that the VRItem object of the spoon checks its related costmaps for a relation between the calculated spoons component and another bowl component. If a relational costmap was found, only the component of the bowl gets returned. The returned component for the bowl is presented in Figure 20(a). This works too, if instead of the spoon a bowl was placed on the kitchen island table and a spoon should be placed as presented in Figure 20(b).



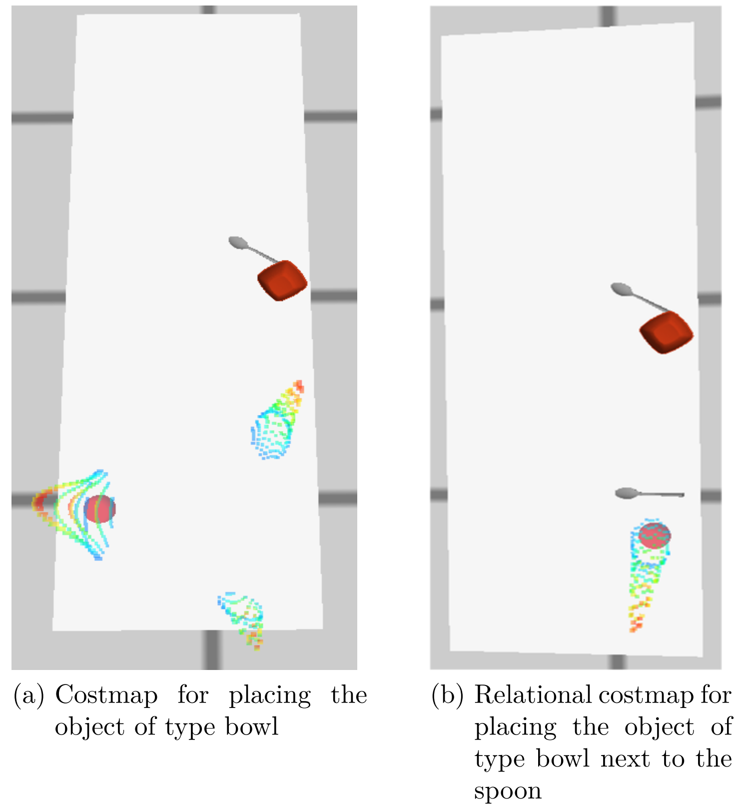
(a) Relational costmap of the bowl after one spoon was placed

(b) Relational costmap of the spoon after one bowl was placed

Figure 20: Visualized relational costmaps of the different object types spoon and bowl

Moreover, costmap_learning recognizes, if the relational costmap is already masked by two different objects. If e. g. a bowl and spoon were already placed next to each other as shown in Figure 21(a), costmap_learning recognizes the covered relational costmap and does not return it. If no other relational costmap can be found, a cut destination costmap of the wanted object will be returned as presented with object bowl in Figure 21(a). But if another relational costmap was found, since e. g. another spoon was placed on kitchen island table as shown in Figure 21(b), costmap_learning

returns the component which is in relation with the placed spoon and has the wanted object type.

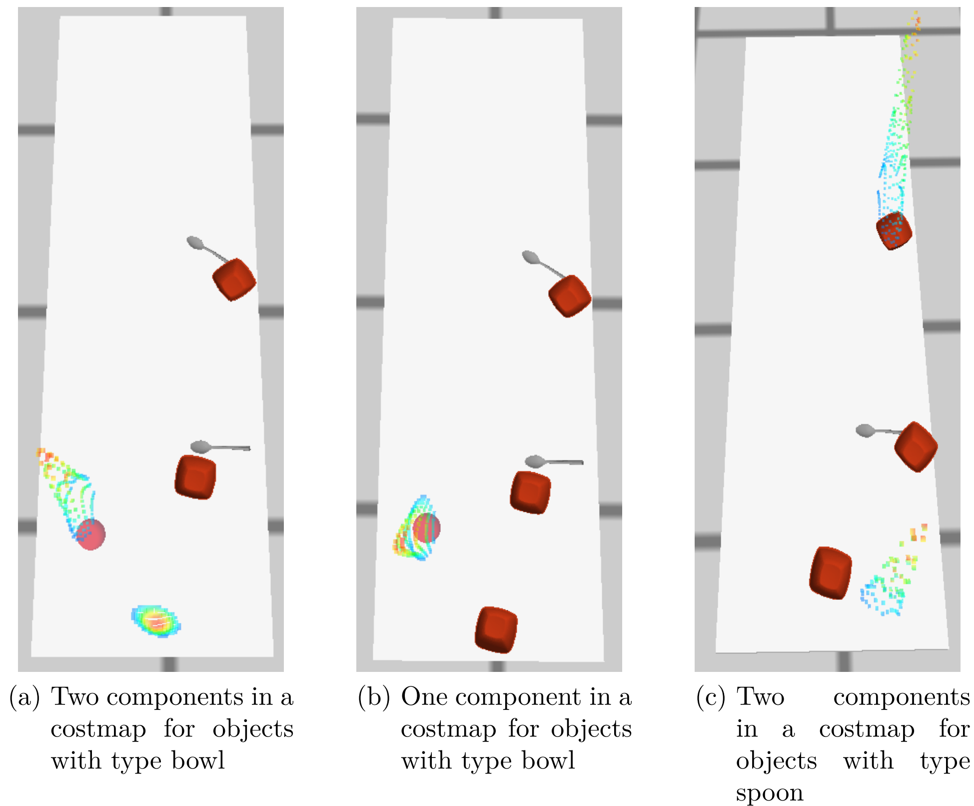


(a) Costmap for placing the object of type bowl

(b) Relational costmap for placing the object of type bowl next to the spoon

Figure 21: Visualized costmap for placing objects of type bowl

The generic procedure returning the correct destination costmaps in costmap_learning can be applied on different kitchen states. The Figures 22(a), 22(b) and 22(c) show more complex use cases for breakfast table settings. Figure 22(a) shows the correct returned destination costmap for a bowl after two bowls and spoons were already placed next to each other. Therefore, the relations between each spoon and bowl get recognized from costmap_learning and thus are not returned. Instead the destination costmap of the bowl returned only components being not masked by the placed bowls. Figure 22(b) shows another use case of a cut destination costmap of the object type bowl. The bowl at the bottom of Figure 22(b) was placed with a sampled pose from the two components visualized in Figure 22(a). Since costmap_learning do not allow reflexive relations, the destination costmap for the bowl returns one component. Lastly, the Figure 22(c) shows, that costmap_learning can return components of multiple relational costmaps too, if there are enough objects placed to which a relation can be established. In particular, the Figure 22(c) shows that two bowls have each no spoon placed next to it. costmap_learning therefore returns both relational components representing the spoon placements. Without relational costmaps, three components for the spoon would be returned. Thus, for breakfast settings the robot must not check, if next to the desired spoon placement a bowl was placed.



(a) Two components in a costmap for objects with type bowl

(b) One component in a costmap for objects with type bowl

(c) Two components in a costmap for objects with type spoon

Figure 22: Visualized costmaps for objects of type bowl and spoon in other table setup

Since the model in costmap_learning did learn the storage placements too, these can be retrieved and visualized too. Figure 23 shows the storage costmap of the spoon. Since the spoon was grasped, after the drawer holding the spoons was opened, the storage costmap visualizes from where the spoons were grasped from after opening the drawer. Another storage costmaps is shown in the Figure 33(c) presenting the storage placements of the bowl. Moreover, the learned orientations allow the robot to grasp the objects accurately too. Figure 34 shows the spoons and bowls storage orientations with visualized sampled poses.

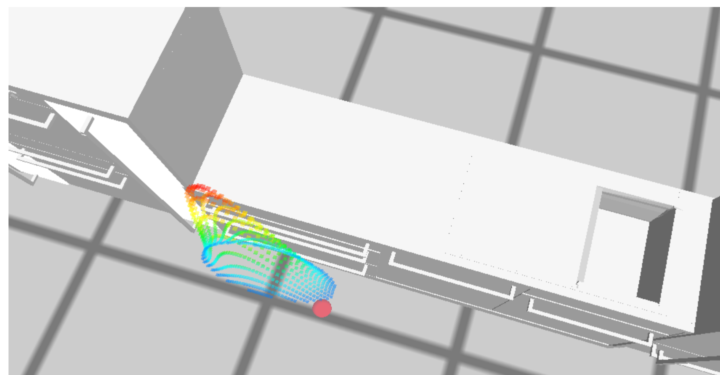


Figure 23: Visualized costmap of the storage placements of the spoons

6 Conclusion

6.1 Summary

To summarize, the built ROS package costmap_learning achieves the goal explained in the Hypothesis 1.2. The robot can use the developed package to store objects like the human did by acquiring the symbolic locations and subsymbolic placements of the different objects. For table settings, it will use the same objects and place them by imitating the human. Therefore, the used models of the object placements encode the commonsense of the human. Moreover, the robot can utilize its knowledge around the environment to get more accurate object placements with relational and cut costmaps to assure robust table settings with faster planning times. This concludes in fast and appropriate breakfast settings for different kitchen setups without the use of any static heuristics.

6.2 Discussion

Although, the implemented system covers the requirements for placements of the objects used in a breakfast scenario, still different problems arise with the current implementation. One existing problem is, that the object coordinates were recorded in the global map frame. If the table would be moved in the Unreal Engine in VR or in the simulation Bullet in CRAM, objects would not be placed accordingly on the kitchen island table. Therefore, the objects coordinates should be recalculated in the base frame of the kitchen island table. The model used in costmap_learning could use these recalculated positions without any adjustments.

Moreover, currently the favorite seating position is not represented in the visualized costmaps as shown in Figure 33(a), since all components have the same weight. Although, the destination placements GMMs inside of costmap_learning did encode favorite seating positions of different items, this parameter was omitted in the distribution returned to CRAM. The same weight of every component allowed during evaluation and debugging better testing results, since objects were placed more often in different components.

Another minor problem is, that placed objects can be pretty close to each other since the object sizes are not modeled. This problem could be reduced by cutting the returned components. Since CRAM checks with the simulation Bullet if during a object placement, collisions with the placed objects occur, stacking objects was always circumvented.

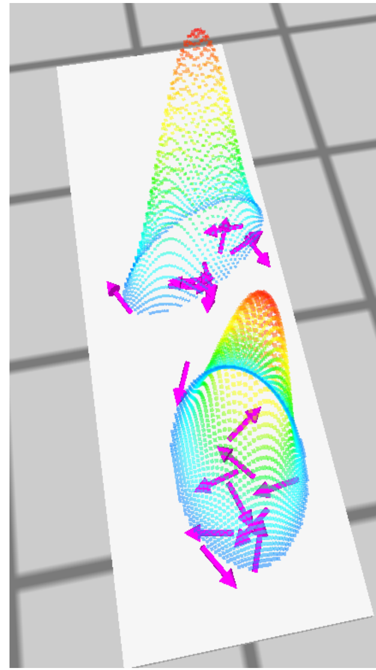
Moreover, relation costmaps can cause problems because they do not represent classes of different objects. If e. g. a plate was placed on the kitchen island table, the relational costmap for a bowl would return a distribution covering the placed plates placements too. This can either, be a wanted behavior if e. g. object should be stackable or should

be specified with different object classes e. g. cutlery or dishes. This problem could be solved by sending not only the placed object types and positions to costmap_learning, but the object classes too. The implementation in costmap_learning needed only small adjustments to allow this new behavior. Additionally, it could be discussed if one-to-one-relations are sufficient enough for table settings or if other multiplicities for relations could be relevant.

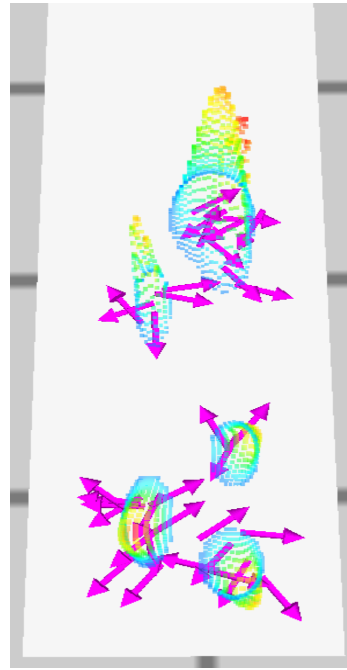
Lastly, clustering problems of some object types do exist, due to the lack of density in the collected data set. Therefore, object placements of e. g. the cup were clustered in just two clusters as represented in Figure 24(a), although clustering in at least five components would represent the points better. Even after the component number was set fixed to six (see Figure 24(c)), since the Silhoutte score did not change much as shown in Figure 35, the orientations of the sampled poses of each component visualized in Figure 24(b) show still a high variances. This does not represent the orientations of the cups from the VR experiments exactly. Moreover, the object cup had the most complex object placements, since I did not choose always the same placement for every seating position[27]. Thus, if the borders[28] of the objects destination placements are set to be closer to each other, much more data is needed to achieve clusters modeling appropriate object placements. At the end, it could still be, that the GMM would cluster cup placements for two different seating positions in one component. Therefore, the number of seating positions should be learned from the learned dishes and cutlery placements, instead of using the silhouette score. The number of seating positions could allow to create a hierarchic structure for object placements, so that object placements could concentrate on specific seating positions instead of the whole table.

[27] If e. g. a bowl was placed, the cups were sometimes placed on top right or on the top left of the bowl.

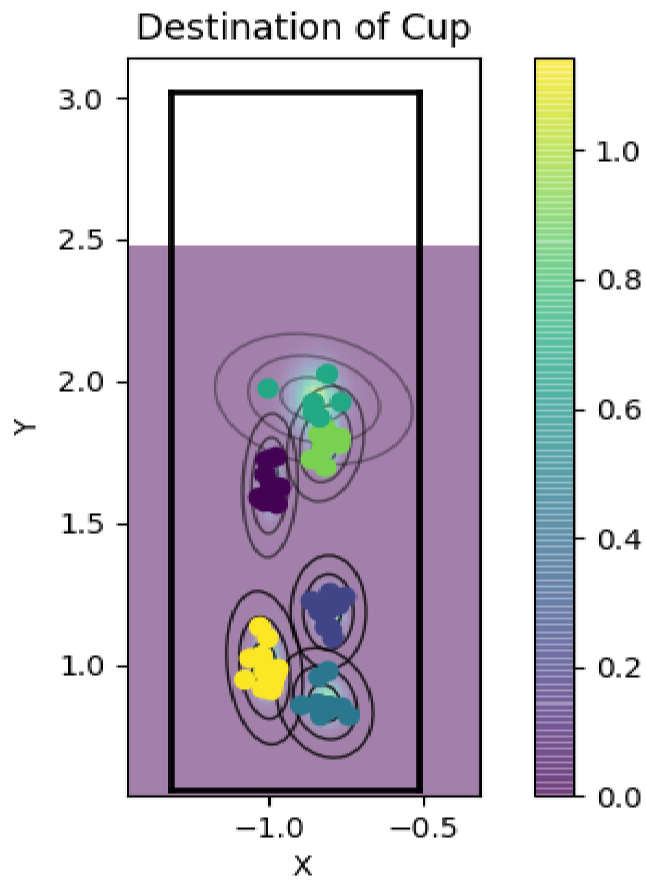
[28] since the GMMs are a continuous model, border denotes in this context the points which depart with $\pm 3\sqrt{\sigma}$ from the mean



(a) Two components in a costmap for object with type cup with sampled poses



(b) Six components in a costmap for object with type cup with sampled poses



(c) Six components in a costmap for object with type cup

Figure 24: Visualized destination costmap for objects of type cup

6.3 Future Work

First, the coordinate frame in which the objects were recorded, should be changed to the base frame of the kitchen link on which the objects were placed. This would allow to move the table resulting in various possible kitchen setups and would make the object placements only depend on the particular table shape and not anymore on its pose in the kitchen. Moreover, different tables could be used in smaller sizes and/or other shapes. With the measurements of the different shaped table, the learned object placements could be transformed for smaller or bigger tables of the same shape type. Additionally, costmap_learning could learn not just the placements of the objects destination or storage, but for other intermediate destinations too. Thus, objects could be placed first e. g. on a tray and then be transported on the tray to the table. The robot could learn where to put the tray for loading it with different objects and where to place it to put these objects to their final position. Moreover, the robot would learn the placements of the used objects on the tray.

Furthermore, it could be investigated, if the order in which the objects were placed could matter for faster table settings and if the robot would perform better. Due to, the limitation that currently one robot arm can only pick one object, other cognitive behaviors could be considered for more complex pick and place tasks with objects.

Another approach to make planning more efficient would be in learning the robot poses for pick and place tasks of the different objects. The robot poses could be clustered to assure more accurate sampled robot poses for placing objects on different seating positions. A requirement for this would be, that the human recording the VR experiments could only move in the range of the robot.

Lastly, the information recorded in the VR data allows for various other imitation learning approaches. The humans arm movements could be used to learn e. g. pouring or cutting motions of particular objects, so that the robot can interact more robust and convenient with household tasks in the kitchen environment.

Bibliography

[1] Tamas Bates, Karinne Ramírez-Amaro, Tetsunari Inamura, and Gordon Cheng. On-line simultaneous learning and recognition of everyday activities from virtual reality performances. pages 3510–3515, 09 2017.

[2] Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoglu, and Georg Bartels. Knowrob 2.0 – a 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *International Conference on Robotics and Automation (ICRA)*, Brisbane, Australia, 2018.

[3] Asa Ben-Hur, David Horn, Hava T. Siegelmann, and Vladimir Vapnik. Support vector clustering. *J. Mach. Learn. Res.*, 2:125–137, March 2002.

[4] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

[5] Erwin Coumans. Bullet: real-time collision detection and multi-physics simulation. `https://github.com/bulletphysics/bullet3`. Accessed: 2020-03-09.

[6] G. Hinton D. Rumelhart and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[7] Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. *CoRR*, abs/1709.04905, 2017.

[8] J Randall Flanagan, Miles C Bowman, and Roland S Johansson. Control strategies in object manipulation tasks. *Current Opinion in Neurobiology*, 16(6):650 – 659, 2006. Motor systems / Neurobiology of behaviour.

[9] David Peel Geoffrey J. McLachlan. *Finite Mixture Models*. Wiley, 1 edition, 1995.

[10] A. Haidu and M. Beetz. Action recognition and interpretation from virtual demonstrations. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2833–2838, Oct 2016.

[11] Andrei Haidu and Michael Beetz. Automated models of human everyday activity based on game and virtual reality technology. In *International Conference on Robotics and Automation (ICRA)*, Montreal, Canada, 2019.

[12] Andrei Haidu, Daniel Beßler, Asil Kaan Bozcuoglu, and Michael Beetz. Knowrob-sim - game engine-enabled knowledge processing towards cognition-enabled robot control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018*, pages 4491–4498, 2018.

[13] Andrei Haidu, Daniel Kohlsdorf, and Michael Beetz. Learning action failure models from interactive physics-based simulations. In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, Hamburg, Germany, 2015.

[14] Alina Hawkin. Towards robots executing observed manipulation activities of humans, 2018.

[15] Dominik Jain, Lorenz Mösenlechner, and Michael Beetz. Equipping robot control programs with first-order probabilistic reasoning capabilities. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation*, ICRA'09, pages 3130–3135, Piscataway, NJ, USA, 2009. IEEE Press.

[16] G. Kazhoyan and M. Beetz. Programming robotic agents with action descriptions. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 103–108, Sep. 2017.

[17] G. Kazhoyan, A. Hawkin, S. Koralewski, and M. Beetz. Learning robust motion parameterization for fetch and place tasks from observing human in virtual environments. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[18] S. Koralewski, G. Kazhoyan, and M. Beetz. Self-specialization of general robot plans based on experience. *IEEE Robotics and Automation Letters*, 4(4):3766–3773, Oct 2019.

[19] Lorenz Mösenlechner. The cognitive robot abstract machine - a framework for cognitive robotics, 2015.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[22] Benjamin Rosman and Subramanian Ramamoorthy. Learning spatial relationships between objects. *The International Journal of Robotics Research*, 30(11):1328–1342, 2011.

[23] S. Schaal and C. G. Atkeson. Learning control in robotics – trajectory-based opitimal control techniques. 17(2):20–29, 2010. clmc.

[24] He Wang, Sören Pirk, Ersin Yumer, Vladimir Kim, Ozan Sener, Srinath Sridhar, and Leonidas Guibas. Learning a generative model for multi-step human-object interactions from videos. In *Computer Graphics Forum*, 2019.

[25] T. Welschehold, C. Dornhege, and W. Burgard. Learning manipulation actions from human demonstrations. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3772–3777, Oct 2016.

Appendix

Figures

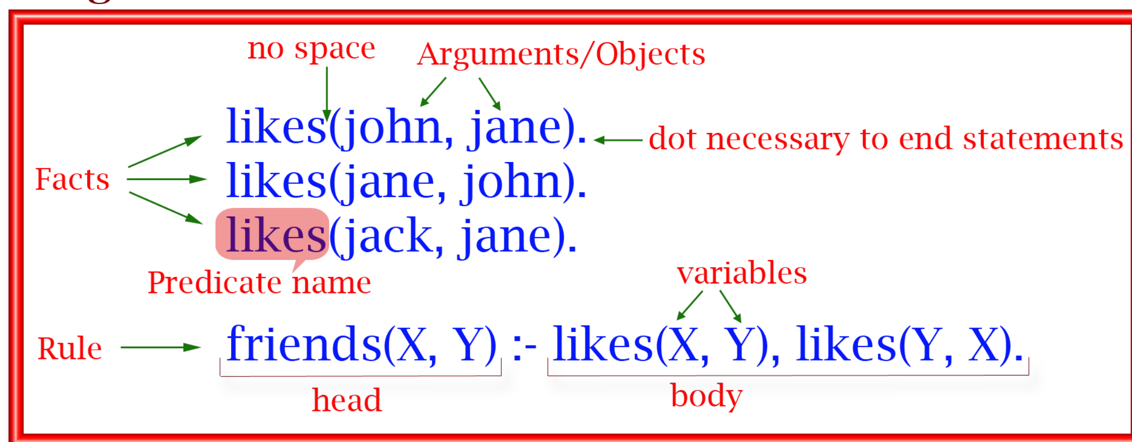


Figure 25: Prolog definition of the rule `friends` with the facts `likes`[29]

```
1   ;; drive
2   (<- (desig:motion-grounding ?desig (drive ?motion))
3         (desig-prop ?desig (:type :driving))
4         (desig-prop ?desig (:speed ?speed))
5         (lisp-fun make-turtle-motion :speed ?speed ?motion))
```

Figure 26: Referencing of a simple motion designator of type `drive`[30]. Prolog variables start with the prefix “?“. In line 3 `desig-prop` checks if the given designator `?desig` contains the key `:type` with the value `:driving`. In line 4 `desig-prop` checks if the given designator `?desig` contains a key `:speed` and writes the value in `?speed`. In line 5 the lisp function `make-turtle-motion` gets called with the arguments `:speed`, the assignment of `?speed` and writes the result in `?motion`. If the facts `desig-prop` and the function call of `make-turtle-motion` returns not nil, the lisp function `drive` with the result of `make-turtle-motion` in `?motion` gets executed.

[30] Author: Vishma Shah, Website: http://athena.ecs.csus.edu/~mei/logicp/prolog/programming-examples.html

[30] Author: Gayane Kazhoyan, Website: http://cram-system.org/tutorials/beginner/motion_designators#defining_inference_rules_for_designators

Figure 27: Cutlery drawer from the kitchen sink area in the Unreal Engine

Figure 28: Cutlery drawer from the kitchen sink area. Figure taken from [11]

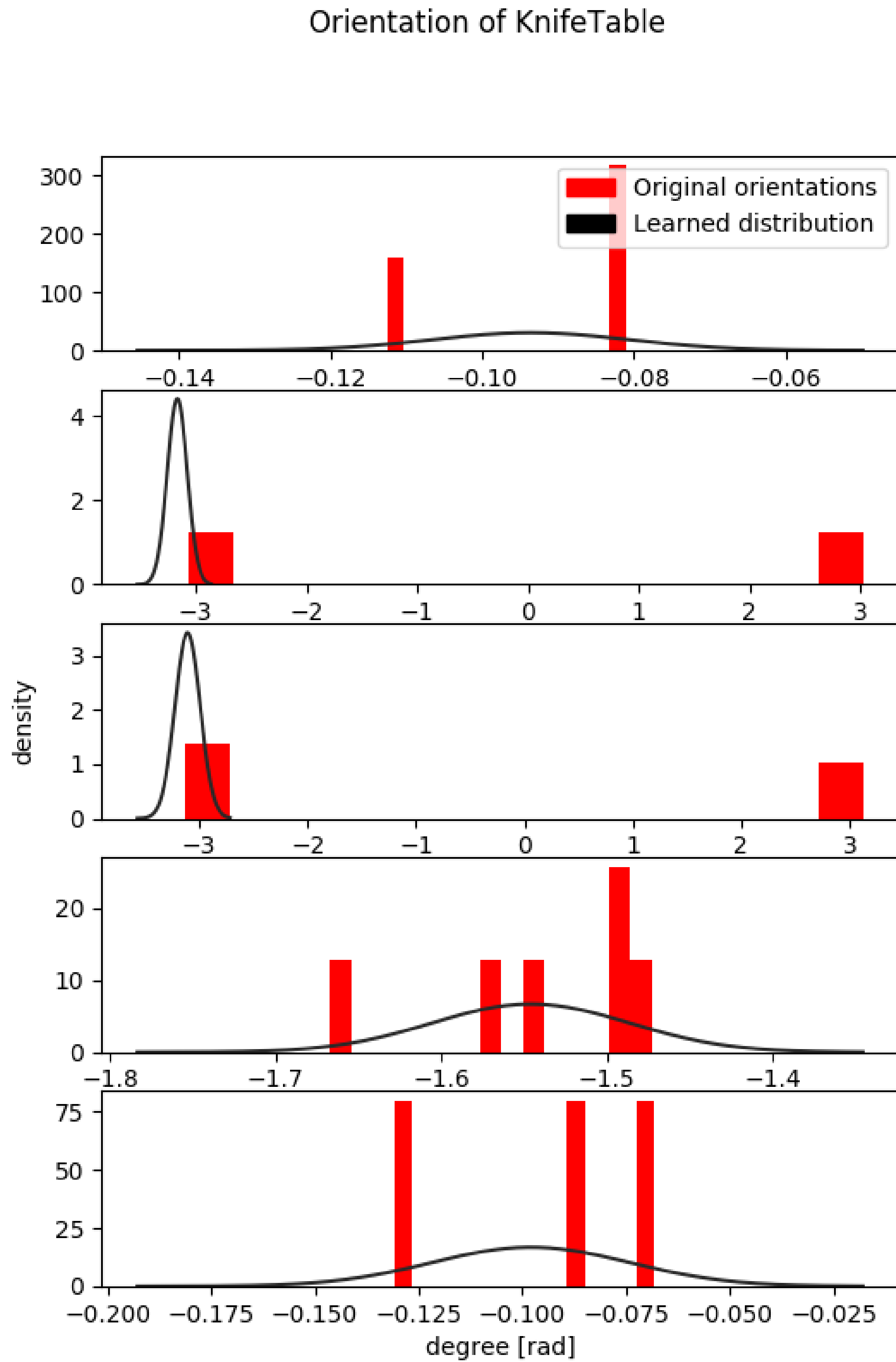


Figure 29: The destination orientation distributions of the object KnifeTable

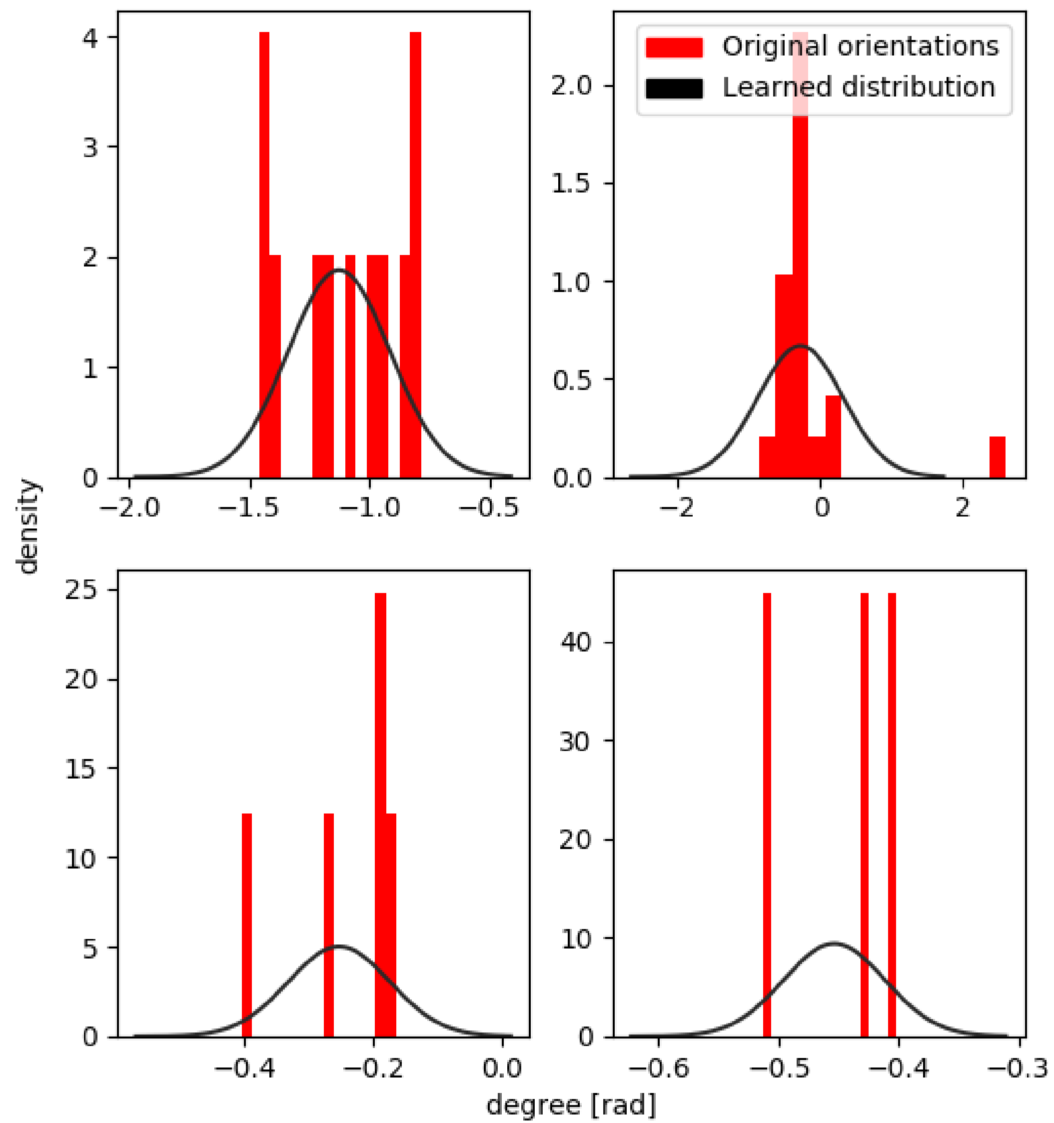


Figure 30: The destination orientation distributions of the object BowlLarge

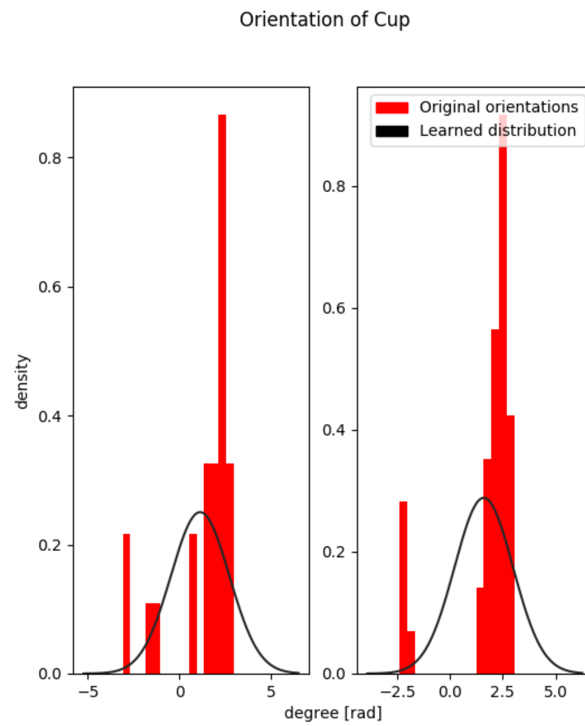


Figure 31: The destination orientation distributions of the object Cup

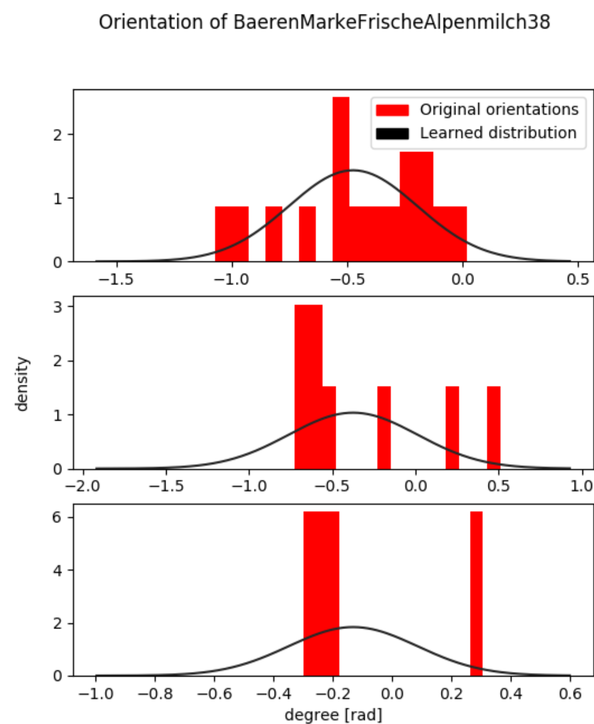
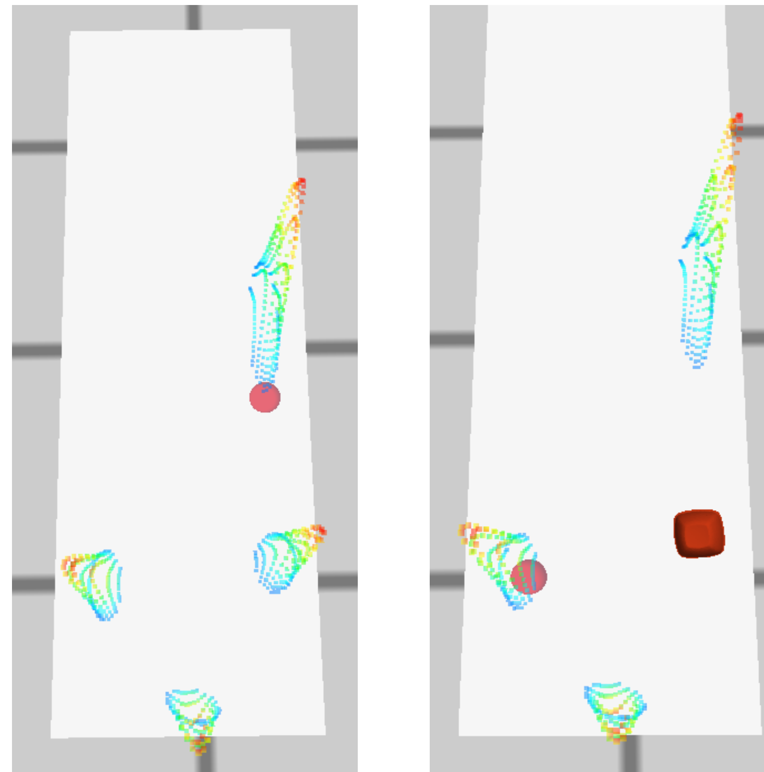
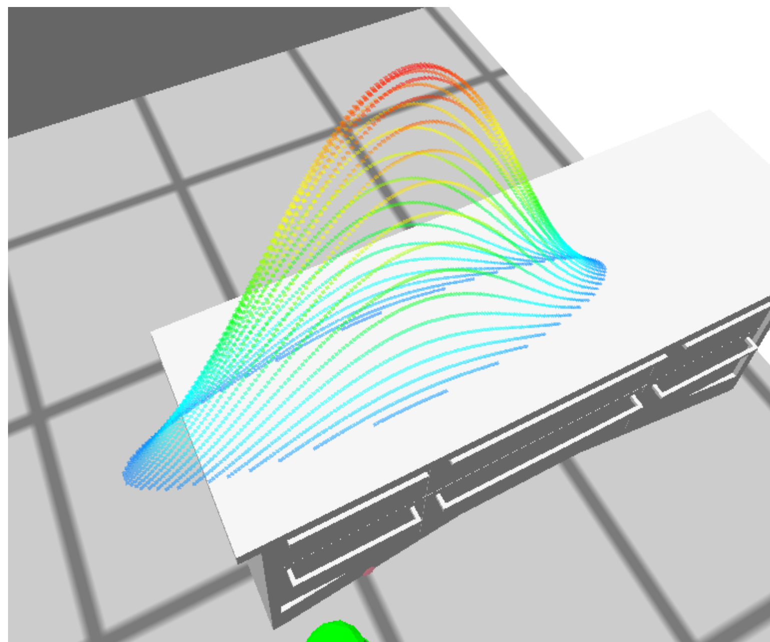


Figure 32: The destination orientation distributions of the object BaerenMarke-FrischeAlpenmilch38



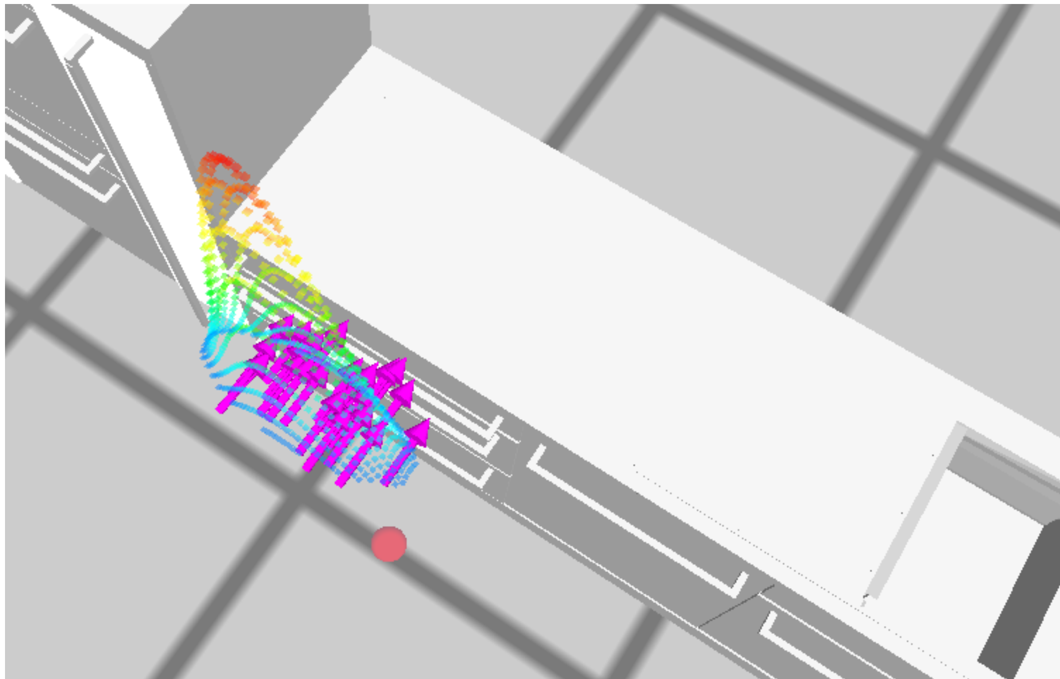
(a) The destination costmap of the objects with type bowl

(b) The destination costmaps of the objects with type bowl after one bowl was already placed

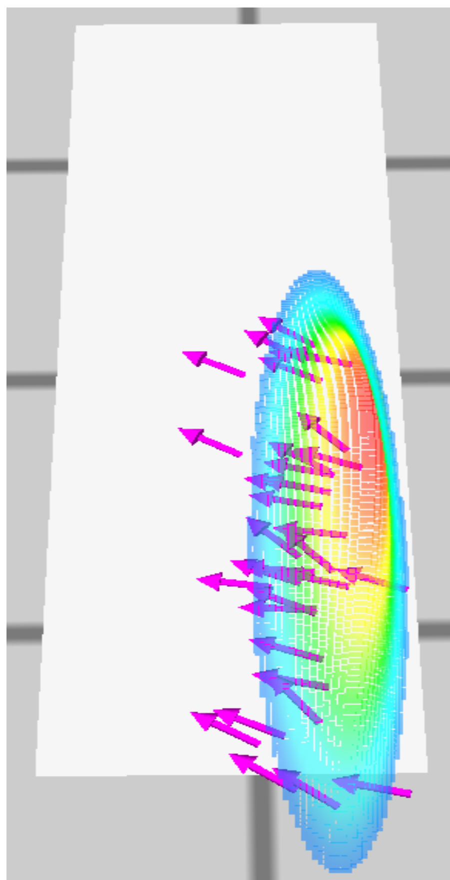


(c) Storage costmap of the objects of type bowl

Figure 33: Visualized costmaps for objects with the object type bowl



(a) The storage orientations of the objects with type spoon



(b) The storage orientations of the objects with type bowl

Figure 34: Visualized storage costmaps visualized with purple arrows for objects of type spoon and bowl

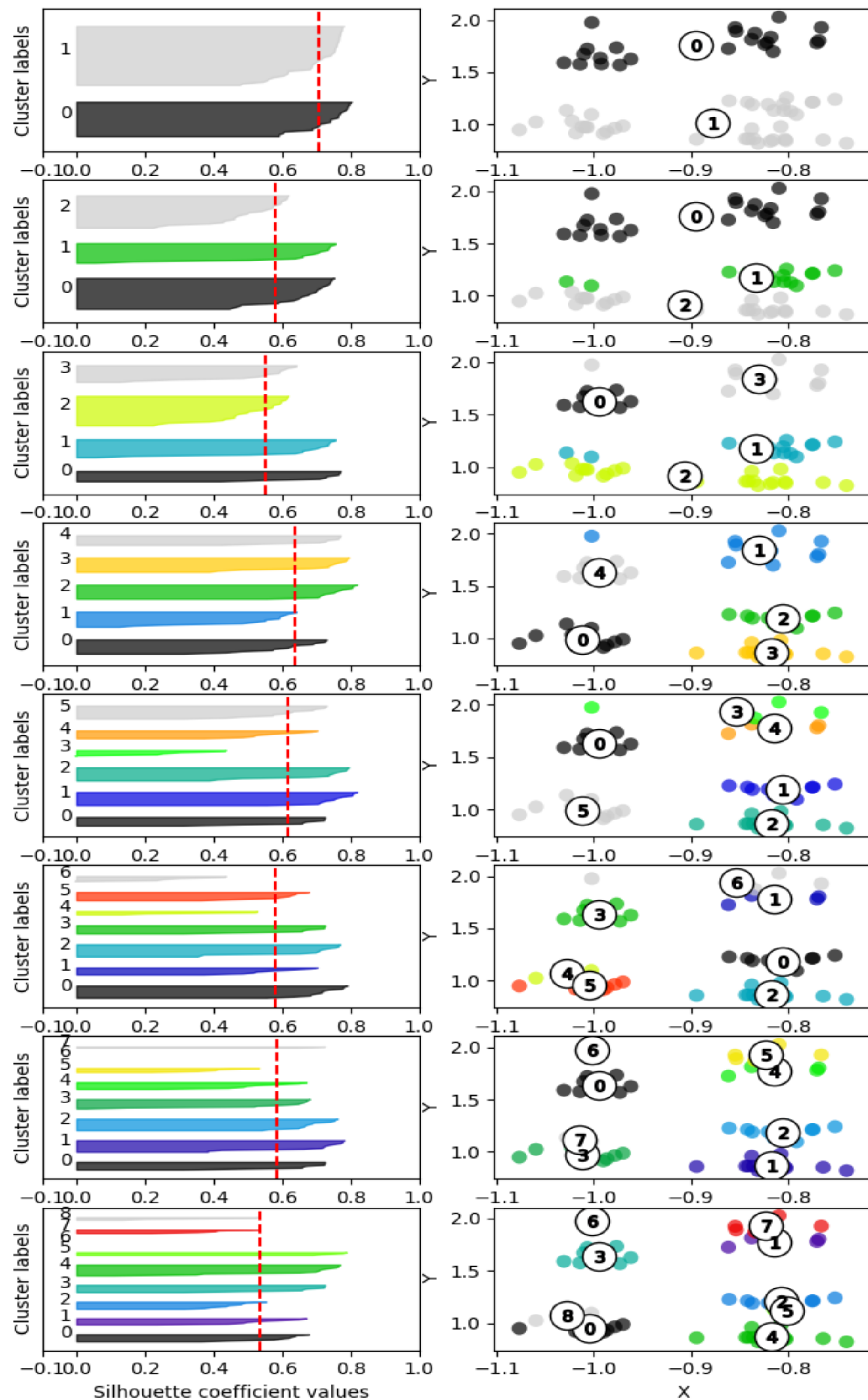


Figure 35: Silhouette scores for clusters quantities of objects with type cup

List of Figures

1 Visibility costmap of the bowl represented as Gaussian distribution . . 20
2 Visibility costmap of bowl represented as Gaussian distribution showing the orientation of the generated/sampled and validated pose 20
3 The pipline from the Unreal Engine until the robot simulation, Figure taken from [14] 21
4 The data flow of the symbolic and subsymbolic placement information acquired from VR experiments 24
5 Simulated kitchen environment showing on the left the **kitchen island** with drawers and on the right the **kitchen sink area**. Left from the **sink area** is some space on the work place and in the left corner is an oven coated from two **pull-out shelves**. The right **pull-out shelf** is opened. The **fridge** is on the right of the **kitchen sink area**. Moreover, this Figure shows a full breakfast setup on the **kitchen island**. 25
6 The timeline of the VR experiment showing the manipulation of a cup and the kitchen 26
7 Real kitchen environment showing on the left the **kitchen island** with drawers and on the right the **kitchen sink area**. Left from the **sink area** is some space on the work place and in the left corner is an oven coated from two **pull-out shelves**. The **fridge** is on the right of the **kitchen sink area**. The photo shows the robot PR2 doing a pick and place task on the **kitchen island**. Location: Laboratory of the Institute of Artificial Intelligence in the University Bremen. Figure taken from [17] 27
8 Stored objects in the fridge (left) and all usable objects on the kitchen island table (right). Figure taken from [11] 27
9 One full_breakfast_setup VR experiment shows how the kitchen island was placed with big bowls, spoons, cups, a plate, knife, fork, milk, cereal box and juice. 29
10 The component diagram showing the services GetCostmap and GetSymbolicLocation of the built system costmap_learning 32
11 The class diagram of the built system costmap_learning 36
12 Visualized Costmap objects of different VRItem objects BowlLarge and SpoonSoup 37
13 Visualized destination costmap objects of the different VRItem objects BowlLarge and SpoonSoup. The used model in the destination costmap was the BayesGaussianMixture from sklearn [20], which created covariances not fitting to the clusters of the points. 39
14 The destination costmaps of BowlLarge and SpoonSoup overlayed . . . 41
15 The destination orientation distributions of the object SpoonSoup . . . 42

16 The destination orientation distributions of the object PlateClassic28 . 43
17 Visualized relational costmap objects between the different VRItem objects BowlLarge and SpoonSoup 46
18 Visualized costmaps of the object type spoon 49
19 More destination costmaps for objects with the types knife, bowl and plate 50
20 Visualized relational costmaps of the different object types spoon and bowl 51
21 Visualized costmap for placing objects of type bowl 52
22 Visualized costmaps for objects of type bowl and spoon in other table setup 53
23 Visualized costmap of the storage placements of the spoons 53
24 Visualized destination costmap for objects of type cup 56
25 Prolog definition of the rule `friends` with the facts `likes`[31] 60
26 Referencing of a simple motion designator of type `drive`[32]. Prolog variables start with the prefix “?“. In line 3 `desig-prop` checks if the given designator `?desig` contains the key `:type` with the value `:driving`. In line 4 `desig-prop` checks if the given designator `?desig` contains a key `:speed` and writes the value in `?speed`. In line 5 the lisp function `make-turtle-motion` gets called with the arguments `:speed`, the assignment of `?speed` and writes the result in `?motion`. If the facts `desig-prop` and the function call of `make-turtle-motion` returns not nil, the lisp function `drive` with the result of `make-turtle-motion` in `?motion` gets executed. 60
27 Cutlery drawer from the kitchen sink area in the Unreal Engine 61
28 Cutlery drawer from the kitchen sink area. Figure taken from [11] . . . 61
29 The destination orientation distributions of the object KnifeTable . . . 62
30 The destination orientation distributions of the object BowlLarge . . . 63
31 The destination orientation distributions of the object Cup 64
32 The destination orientation distributions of the object BaerenMarke-FrischeAlpenmilch38 64
33 Visualized costmaps for objects with the object type bowl 65
34 Visualized storage costmaps visualized with purple arrows for objects of type spoon and bowl 66
35 Silhouette scores for clusters quantities of objects with type cup 67

List of Tables

1 Collected and exported episodes 28
2 Some samples from the exported episodes 31

Listings

1 Base information to query VR data in KnowRob with Prolog 13
2 Example of a location designator 19

Acronyms

AI .... Artificial Intelligence
CPL .... CRAM Plan Language
CRAM .... Cognitive Robotic Abstract Machine
CSV .... Comma-Seperated Values
EM .... Expectation-Maximization
FO .... First-Order logic
GMM .... Gaussian Mixture Model
GUI .... Graphical User Interface
JSON .... JavaScript Object Notation
KnowRob .... Knowledge processing for Robots
NB .... Naive Bayes
OS .... Operating System
OWL .... Web Ontology Language
RGB .... Red Green Blue
RNN .... Recurrent Neural Network
RobCoG .... Robot Commonsense Games
ROS .... Robot Operating System
SVM .... Support Vector Machine
VR .... Virtual Reality