Universitat Politècnica de Catalunya

Master Thesis

Master's degree in Automatic Control and Robotics

Task planning for table clearing of cluttered objects

Author:
Nicola Covallero

Director:
Guillem Alenyà Ribas
Co-director:
David Martínez Martínez
Tutor:
Guillem Alenyà Ribas

June 2016

Abstract

Manipulation planning is a field of study with increasing interest, it merges manipulation skills with an artificial intelligence system that is able to find the optimal sequence of actions in order to solve manipulation problems. Those problems are complex since they involve a mixture of symbolic planning and geometric planning.

In this thesis we present a planning system for clearing a table of cluttered objects, which completely reasons at a symbolic level postponing the reasoning with geometric constraints through backtracking. The task we face is a complex task that should include geometric information, such as the objects positions, during the planning step.

We show that the planning system developed is able to handle efficiently the task with symbolic predicates.

The main contribution of this thesis is a planning system able to solve a wider variety of scenarios for clearing a table of cluttered objects. Grasping actions alone are not enough, sometimes pushing actions are needed in order to move an object in a pose in which it can be grabbed. The planner here presented is able to understand when, and where, to push an object and why it is not possible to grasp it.

This work shows how some geometric problems can be efficiently handled by reasoning at an abstract level through symbolic predicates when such predicates are chosen correctly. The advantage of this kind of planning system is a reduction in computation time and also an easier implementation.

Contents

1. Introduction

Robotic manipulation of objects is an increasing field of research which has struggled researches from many years ago. In several industrial environments robots can be easily programmed considering the objects to be known a priori, i.e. the manipulation is always the same, and usually robot operations avoid to deal in cluttered scene, but this situation needs a workspace designed in a manner to provide to the robot a non cluttered scene. But there are situations in which robots with enhanced intelligence can be useful. An example in which a robot could face a cluttered scenario is the one of the *Amazon Picking Challenge* [8], which has the objective to provide a challenge problem to the robotics research community that involves integrating the state of the art in object perception, motion planning, grasp planning, and task planning to manipulate real-world items in industrial settings such ones human operators face in Amazon's warehouse. Joey Durham from Amazon Robotics describes the challenges of this competition with the following sentence:

> "A robot for picking purposes must possess a lot of skills: The selected item must be identified, handled and deposited correctly and safely. This requires a certain level of visual, tactile and haptic perceptive skills and great handling capabilities."

In this thesis we are going to investigate a simple approach for a complex manipulation problem. Our approach tries to replicate the human reasoning during the manipulation of cluttered objects in table clearing tasks. The objective is removing all the objects onto a table considering that their poses could limit the manipulation actions.

This thesis has been developed in the **Institut de Robòtica i Informàtica Industrial** (IRI) in the Perception and Manipulation laboratory with the supervision of *Guillem Alenyá Ribas* as director and *David Martinéz Martinéz* as co-director.

This work is structured as follow. First an introduction to the problem we want to solve is presented in the Chapter 1. Then in Chapter 2 a review of the current state of the art in manipulation planning is done. The planner developed will be commented in Chapter **??** and the implemented algorithm to compute the predicates and control the robot will be commented in Chapter **??**. In Chapter 4 the softwares employed for the implementation of the code will be presented as well the algorithm's structure. Finally some experiments and conclusions about this work are discussed in Chapter 6 and 7 respectively.

In this chapter first an introduction to the problem and the approach adopted is commented, then the experimental set up is presented since it will be useful to understand some techniques and choices for the algorithm which will be discussed in next chapters.

1.1. Problem Approach

In this section the approach to solve the planning problem is described. The strategy to solve the problem is inspired to a human-like solution. A human would solve such task with three main actions: grasp an object, push it or drag it. When it is not possible to grasp an

object, because other objects hinder the human to put the hands in the proper way to grasp the desired one, he/she has to interact with the scene in order to put the desired object in a pose where it can be grasped.

Dragging is a very complex action which will require the robot to lay the end effector on the top of an object. This is a very hard action which would imply an implementation of a reliable controller, moreover the goal is not only to move the object but also not to ruin it.

Pushing action is an action easier than dragging since it only requires to put the end effector in a certain pose and then push the object by moving the end effector. However the pushing action is complex since to push an object making it following a certain path the pushing pose has to be chosen taking into account the shape of the object and a controller would be needed in order to correct the pose along the path. Despite this we handled to perform that action in a easy way by considering only objects with basic shapes.

For these considerations the main actions the robot has to use in order to interact with the objects are grasping and pushing. Grasping is the most important action since it lets to take an object from the pile of objects and drop it somewhere, for instance into a bin, clearing in this way the table. There exist different works facing the same task by focusing only in grasping [15] [39]. The pushing becomes useful considering the problem that two adjacent objects could not be grasped if they are so close such that the robot's gripper, when attempting to grasp an object, is going to collide with the adjacent one, making the object of interest ungraspable. From this consideration is necessary the pushing action, in order to separate adjacent objects which could mutually exclude themselves to be grasped.

Being observant to the philosophy of a human-inspired solution, the manipulation of the objects is intended in manner to avoid to interact with more than one object per action. That is, when we push, or grasp, an object only that object will change its pose.

The robot's intelligence is enhanced by a planning system which will return a sequence of actions in order to achieve the goal to clear the table. The robot will reason totally in an abstraction level by considering only symbolic predicates which will be obtained by hand-built mapping functions making simpler the reasoning step. It is important noting that this problem should includes important geometric information which are very useful for the planner, such as how much to push an object, in what direction and the inverse kinematic of the robot. This geometric information are translated into symbolic ones relaxing the planning phase, although these are not able to catch all the geometric information. Then through backtracking the robot will evaluate if the returned plan can be actually executed by the robot (i.e. evaluate if the action to execute involves the robot to move inside its working space), if not it updates the predicates and replans. We assume the world is perfectly known and the actions are not reliable. To overcome this the planner replans after the execution of each action. To detect the objects the Microsoft Kinect sensor is used as vision system and a geometry based segmentation is used to segment the objects in the scene.

1.2. Set Up

In order to make easier the understanding of the approach used, the set up of the environment the robot will work in is here presented.

The robot used is a Barret WAM arm, which is a 7 degree of freedom (DoF) manipulator arm shown in Figure 1.1a. The WAM is noteworthy because it does not use any gears for manipulating the joints, but cable drives, so there are no backlash problems and it is both fast and stiff. Cable drives permit low friction and ripple-free torque transmission from the actuator to the joints. To detect the objects a Kinect camera, a RGB-D sensor, is employed (Figure 1.1b).

(a) Barrett WAM arm **(b)** Microsoft Kinect sensor

Figure 1.1: Robot and vision sensor.

To manipulate the objects the manipulation skills of the robot are enhanced by a gripper designed in the IRI institute and actuated by Dynamixel motors. Such a gripper is depicted in Figure 1.2 from several point of views. Its closing width[1] is 3 centimetres while the opening width[2] is of 8 centimetres, therefore we are constrained to grasp objects with a width in the range $[3 \div 8]cm$.

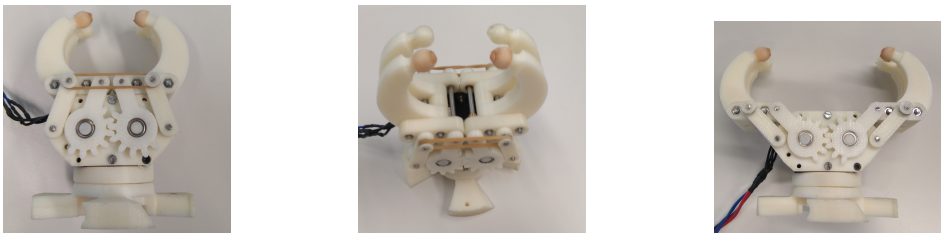


Figure 1.2: Gripper used for the experiments

For the task planner, as the reader will see in Chapters **??** and **??**, the model of the gripper will be an important resource in order to compute the predicates.

The gripper will be modelled measuring some principal elements such as: finger's width and height, gripper's width, height and deep, closing and opening width. The modelling procedure is depicted in Figure 1.4. The resulting model is a simple triangle mesh which includes all the important geometric information of the gripper. Such a simple model allows

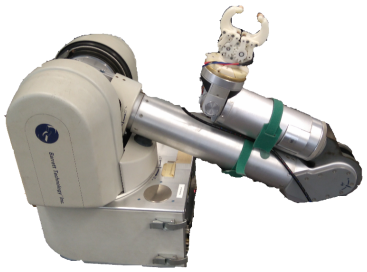
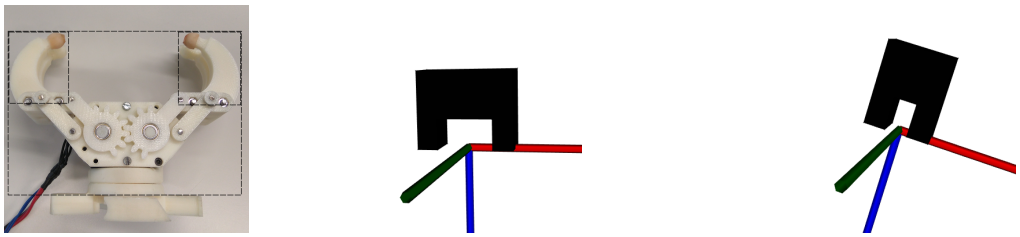


Figure 1.3: Gripper and WAM.

the collision algorithm commented in Chapter **??** to check for collision in just few millisecond. A more detailed and complex model would only lead as benefit a higher precision, but for this kind of task an extremely accurate precision is not needed, and it will slow down the algorithm because the collision checking procedure would be more computationally ex-

[1]Distance between the fingers when the gripper is closed.
[2]Distance between the fingers when the gripper is open.

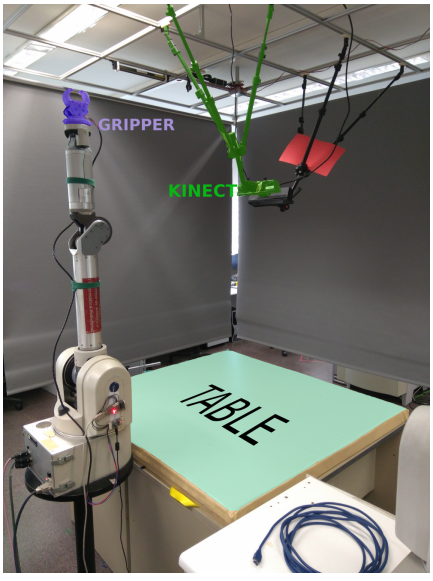
pensive. The gripper will be mounted in the end effector of the robot as shown in Figure 1.3.



Elements measured Opened gripper mesh model Closed gripper mesh model

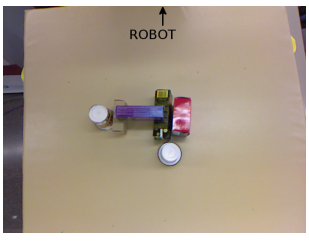
Figure 1.4: At the left the principal elements measured are highlighted for the opened gripper model. The gripper mesh model is here shown in the PCL visualizer. The red, green and blue axis are respectively the x, y and z axis.

The scenario the robot is going to work in is composed of a table and the objects will lay on top of it. In Figure 1.5a the main elements of the set up are highlighted. The WAM arm's base is in a fixed position with respect the table and the Kinect camera is located on top of the table pointing down. The Kinect is calibrated and the fixed transformation between the Kinect's frames and the base frame of the robot is known, so all the points measured by the Kinect can be expressed in coordinates with respect the robot's base frame. An example of a cluttered scene the robot is going to deal with in this thesis is depicted in Figure 1.5b, and the same scene seen by the Kinect is shown in Figure 1.5c. With respect the kinect's view the robot is located at the top side of the image, outside the field of view of the Kinect, in order to avoid to detect the arm as an object. If the robot arm would be present in the depth image we should apply some algorithms to identify the robot arm, which could also produce important occlusions, that is hide some objects from the Kinect's view. In order to avoid all this, the depth images considered are the ones provided by the Kinect when the robot is no more in the Kinect's view (these poses are known a priori).



(a) Principal elements of the experimental set up.

(b) Example of a cluttered scene the robot is going to work with.



(c) Kinect's view.

Figure 1.5: Experimental set up, and an example of a cluttered scene the robot is going to interact with.

2. State of the art in manipulation planning

Many manipulation planning approaches (see [25] for an overview) assume that the task can be treated as a geometric problem, with the goal to place the objects in their desired positions. Planning is essentially done with a mixture of symbolic and geometric states, this requires a set of symbolic predicates that correspond to geometric relationships. Cause generating geometric paths is very time consuming, these hybrid planners could be too slow to be used in real applications.

Dogar and Srinivasa [12] proposed a framework for planning in cluttered scenes using a library of actions inspired by human strategies. They designed a new planner that decides which objects to move, the order to move them, where to move them, and it chooses the manipulation actions to use on these objects, and it accounts for the uncertainty in the environment all through this process. The planning system first attempts to grasp the goal object, than if it is not possible it identifies what is the object that obstacles the action, then such an object is added to a list of the objects that have to be moved. They move those objects in whatever position such that makes the goal feasible. Their work is the most similar to our, but their planning system cannot be directly applied to a table clearing task. The goal is a single object at a time, then to grasp another object they need to replan. The planning system proposed in this thesis is able to perform more complex plans.

A recent alternative proposed by Mösenlechner and Beetz [29] is to specify goals symbolically but evaluate the plan geometrically. The idea is to use a high-fidelity physics simulation to predict the effects of actions and a hand-built mapping from geometric to symbolic states. Planning is conducted by a forward search, and the effects of actions determined by simulating them, and then using the mapping to update the symbolic state. Their method is promising, by the way the integration of a simulator into the planning system is not straightforward since it requires to know the physic of the manipulated objects. Moreover the authors didn't tested their planning system with a complex scene like the one faced in this thesis. Our planning system doesn't use any simulator, instead it uses a simple prediction algorithm to understand how the objects should be manipulated, leading to an easy implementation and a fast execution in the planning step.

In [7] the authors address a problem similar to the one of this thesis. The authors coupled together pushing and grasping actions for a table manipulation task. They use the concept of reachability [40] for each kind of action to quickly exclude impossible poses of the gripper of the planning stage, creating a reliable plan suitable for real-time operation. The authors model the planning problem through MDP (Markov Decision Process), discrediting the world in grid cells and assigning to each one a push and grasp vector defined by the reachability concept. Unfortunately the authors were not able to test the grasp skill with their planning system. Our approach skips the modelling of the probability by considering to replan after each action execution, this eases our planning phase. Moreover, while their method needs to be adapted to each robot, to built a reachability map, our method can be

directly integrated in any robotic manipulator.

Machine learning techniques also have been applied to task planning. To plan complex manipulation tasks, robots need to reason on a high level. Symbolic planning, however, requires knowledge about the preconditions and effects of the individual actions. In [3] the authors proposed a practical approach to learn manipulation skills, including preconditions and effects, based on teacher demonstrations. With few demonstration the authors proposed a method to teach the robot the preconditions and effects of individual actions. Their method furthermore enables the robot to combine the learned actions by means of planning to solve new tasks that are more complicated than the learned individual actions. Although this work looks promising the problem of this approach is that it moves the problem from identifying the right predicates to describe the problem to identifying the right features that let a correct learning of the predicates.

In [11] Dearden and Burbridge proposed an approach for planning robotic manipulation tasks which uses a learned mapping between geometric states and logical predicates. The planner first plans symbolically, then applies the mapping to generate geometric positions that are then sent to a path planner. They try to fit a probability distribution function to the geometric states to map them to a symbolic state. This work sounds promising but the cluttered scene this thesis is going to treat is quite complex to be treated with such a method. Moreover the same authors applied that technique [10] for a simple task which involves putting some cups (singulated between themselves) onto a tray. Although this approach does not involve any hand built mapping function and the results are good, their approach is very time consuming for real applications. The hand-built function still have a good benefit in terms of computational cost, although they need to be adapted to the problem. Despite this also learning the predicates requires to be adapted to the problem.

Compared to the state of the art this thesis proposes a planning system for table clearing tasks of cluttered objects, it plans completely at a symbolic level, efficiently and it is low time consuming (the time to get a plan is usually less than 0.5 seconds). The fast computation of the plan is able to make this system suitable for replanning, overcoming in this way the lack of probability and geometric information. Moreover no other works have been found that face the table clearing of cluttered objects problem, such as the one in Figure 1.5b. It will be shown that the lack of geometric constraints includes some limitations to the system but it plans still efficiently.

3. Planning system

In this chapter the general framework adopted is discussed, proposing a suitable task planning system. After the review of the current state of the art of task planners, a proper planner is chosen and then a suitable description to the table clearing problem is discussed.

3.1. Task Planners Review

To choose the proper planner for the task we evaluated three main categories of planners:

1. classical planners
2. hierarchical planners
3. probabilistic planners

Classical planners are characterized by environments which are fully observable, deterministic, finite and static (changes happen only when the agent acts) and discrete (in time, actions, objects...) [35]. A very well known classic planner is the Fast Downward planner [16].

Hierarchical planning, also called *Hierarchical Task Network*(HTN), works in a similar way to how it is believed that human planning works [27]. It is based on a reduction of the problem. The planner recursively decomposes tasks into subtasks, stopping when it reaches primitive tasks that can be performed directly by planning operators. This kind of planner needs to have a set of methods, where each method is a schema for decomposing a particular kind of task into a set of subtasks. For this kind of planning technique a well known planner is SHOP [30].

Probabilistic planning is a planning technique which considers that the environment is not deterministic but probabilistic. So the actions have a certain probability to obtain a certain state, and given an initial and final state the planner finds the solution path with the highest reward, which depends also on the probability. A well known example on which this kind of planners are build on is the Markov Decision Process. In this category two probabilistic planners that perform good in planning competitions are Gourmand [24] and PROST [22].

3.2. Planner

The problem involves a big amount of uncertainty due to the interaction of the robot with the environment. When the robot will interact with the objects it is very hard to predict correctly the position of the object after the execution, that is the next state, this is a crucial problem which should be considered. With a probabilistic planner, the planner will take into account what object has been moved and it will update the state obtaining a set

of states, each one with a certain probability. The probability also has to be modelled and it is highly depending on the form of the object.

Martinez et al. [28] faced the problem of cleaning a surface entirely by probabilistic symbolic planning. The problem they wanted to solve was characterized by a strong probability factor and a wrong effect of the action would have lead to replan and therefore to slow down the system. Another way to face the problem of probability is replanning after each executed action **cite Tomas Lozano**, or whenever the current state deviates from the expected one, generating a new trajectory from the current state to the goal. The plan's actions are considered deterministic, and the only useful action is actually the first one of the plan, after its execution the system replans again. Little et al. discussed in [26] the problem of when is more useful the probabilistic planning with respect a simple replanning system.

They defined a planning problem *probabilistic interesting* if dead ends can be avoided, exist multiple goal trajectories and there is at least one pair of distinct goal trajectories, τ and $\tau^{'}$ that share a common sequence of outcomes for the first $n-1$ outcomes, and where τ_n and $\tau_n^{'}$ are distinct outcomes of the same action.

They assert that unless a probabilistic planning problem satisfies all of the conditions to be *probabilistic interesting* then it is inevitable that a well-written replanner will outperform a well-written probabilistic planner. Moreover the authors do not negate the possibility that a deterministic replanner could perform optimally even for probabilistically interesting planning problems.

To conclude, a replanner makes more probabilistic problems practically solvable. In the other hand it suffers to be less robust than a probabilistic one.

Taking into account such considerations and that, except for rare cases, our planning problem is not *probabilistic interesting*, the problem has been thought to be solved thanks to a deterministic replanner.

A hierarchical planner would be a good choice if the problem presents some hierarchies, for instance in the case the goal is to clear several tables. Since the problem is about cleaning a single table the problem can be easily modelled through a classic planner and it is straightforward.

The planner chosen to develop the work is the **Fast Downward** planner [16], a very well know classic one. This planner is feature-wise complete, stable and fast in solving planning problem.

3.3. Symbolic Predicates

Once the planner has been chosen, the planning problem has to be formulated accordingly to the capabilities of the planner. Since we chosen a classic one, the problem is deterministic and it is generally formulated as a 6-tuple $\Pi = \langle S^d, s_o^d, G^d, A^d, T^d, c^d \rangle$ [26], where:

- S^d is a finite set of states;
- $s_o^d \in S^d$ is an initial state;
- $G^d \in S^d$ is a goal state;
- $A^d(s)$ is a set of applicable actions for each $s \in S^d$;
- $T^d(a, s) \in S^d$ is a deterministic transition function for all actions $a \in A^d(s)$ and states $s \in S^d$;
- $c^d(a)$ is the cost to apply action a.

The plans, or trajectories, τ_i are sequences of actions applicable from the initial state until the goal state. The cost of a trajectory $C(\tau_i)$ is the sum of the cost of the actions of the trajectory $C(\tau_i) = \sum_{a \in \tau} c^d(a)$. The optimal solution is the solution with less cost: $\tau^* = \min_{\tau_i} c(\tau_i)$. In this work we did not specify any cost to the actions, so the planner will return as solution the one with less number of actions.

The *Fast Downward* planner needs the problem to be formulated in the common format of PDDL (*Problem Domain Description Language*) [2]. In this section the symbolic predicates that have been considered in order to solve the problem are described.

The task this thesis faces is a common task done by humans, who think in order to find a feasible sequence of actions. Such a sequence is normally composed of actions that avoid the collision between the manipulated objects and the other ones, whenever possible. To do this we, as humans, think on what is going to happen if we manipulate an object in a certain way. The design of the problem has been inspired by such a reasoning way and therefore we will handle carefully the objects avoiding collisions between them. The reasoning part is entrusted to the planner, which needs a coherent problem description through symbolic predicates.

As described in the introduction, the system will be able to perform two types of actions: **pushing** and **grasping**. Grasping action is a necessary action in order to grasp an object and drop it into a bin, while the pushing action is an auxiliary action which has the aim to move an object in a pose that do not obstacle the solution of the problem. The pushing action is said to be auxiliary because it is not strictly necessary to solve every kind of problem, depending on the cluttered scene the grasping action could be enough. The blending of these two actions makes wider the range of problem this planner can solve.

The symbolic predicates are designed accordingly to the available actions trying to answer the following questions:

- When can an object be grasped?
- When can a object be pushed? In which direction?

Answering these questions the following predicates are defined:

- **grasped**: `(grasped o1)` means that object `o1` has been grasped and dropped into the bin. The goal is reached when all the object have been grasped.
- **on**:`(on o1 o2)` means that object `o1` stands on top of object `o2`. This predicate is defined since we don't want to interact with an object that has objects on top of itself. If we grasp it, the object above will likely fall corrupting in this way the scene. That behaviour is undesired since a human would not grasp the bottom object without first grabbing the one on the top. Similarly for the pushing action, when an object with objects on top of itself is pushed, they could fall or collide with other objects. Vice versa if it was on top of other objects.
- **block_grasp**: `(block_grasp o1 o2)` means that object `o1` obstacles object `o2` to be grasped. Once we are sure that an object has no objects on top of it we have to check if it can be grasped, that is if the gripper will collide with some objects attempting to grasp the desired one. With this predicate the planner knows that the robot has first to interact with those objects before to grasp the desired one.
- **block_dir1**, **block_dir2**, **block_dir3**, **block_dir4**: `(block_dir1 o1 o2)` means that object `o1` obstacles object `o2` to be moved along direction 1. We will consider 4 possible pushing directions(defined in Chapter 4) per object. Being observant to the philosophy of human-inspired actions, we avoid collisions when we push an object, to

do that we translate the object along the pushing direction, for a certain length, and check for collision. Moreover, to push an object the end effector has to be put in the opposite side with respect the pushing direction, so an object cannot be pushed along a certain direction even in the case the gripper collides with an object. Therefore if an object cannot be moved along a certain direction because it would collide with some objects, or the gripper would collide, those have to be moved or grasped.

- **ik_unfeasible_dir1**, **ik_unfeasible_dir2**, **ik_unfeasible_dir3**, **ik_unfeasible_dir4**, **ik_unfeasible_grasp**: `(ik_unfeasible_dir1 o1)` means that the inverse kinematic to push the object `o1` along direction 1 has no solution. Similarly for the other actions. This predicates are added in order to consider the geometrical constraints regarding the working space of the robot.

Grasping Action The **preconditions** to grasp a certain object are:

- no object stands on top of it,
- no object collides with the gripper attempting to grab the desired object,
- the inverse kinematic has solution.

The **effects** of the grasping action are:

- the grasped object is dropped into the bin,
- the grasped object no more blocks other object to be pushed or grasped,
- if the grasped object was on top of other ones, it is no more on top of them.

Pushing Action The **preconditions** to push a certain object, along a certain direction, are:

- no object stands on top of it,
- the manipulated object is not on top of other objects,
- no object collides with the manipulated one, or with the gripper, when pushed,
- the inverse kinematic has solution.

In particular we defined 4 pushing action, one per pushing direction. The symbolic planner is not able to capture all the geometrical information of the problem through symbolic predicates, therefore it is not able to predict the future position of the manipulated object, and so the future states. This problem was handled by considering the object to be pushed far enough in such a way it is singulated from the other ones. Therefore the **effects** of this action are:

- the manipulated object no more blocks other objects to be pushed or grasped,
- the other objects no more block the manipulated object to be pushed or grasped.

Backtracking The geometrical constraints related to the inverse kinematic of the robot have been added in the problem description but their computation is quite expensive. Moreover, as the reader will see in Chapter 4, the actions are defined by more poses, so the inverse kinematic is even more expensive, usually it is a matter of 2 or 3 seconds per action. Computing it for each possible action (we have 5 actions in total, one grasping action and 4 pushing actions) for each object would make the computation of the predicates too expensive making the planning system quite slow. Usually the objects are inside the working space of the robot and the computation of the `ik_unfeasible` predicates is usually unnecessary. To overcome this we used the **backtracking** technique[5]. Backtracking is based on updating the initial state, used to get the current plan, considering the geometrical constraints, and plan again with the updated initial state obtaining in this way a different plan that could be feasible. The obtained plan will still be evaluated and backtracked until a feasible plan is obtained or there exists no plan. Despite this, planning symbolically is very fast therefore replanning several times is not a problem. With this method the inverse kinematic will be solved only for the action we want to execute, the first one of the plan, and no time is wasted in computing the inverse kinematic for unnecessary actions. If executing the first plan's action is not possible the equivalent `ik_unfeasible` predicate is updated. The pseudo algorithm to get a plan is shown in Algorithm 1.

Algorithm 1 Planning procedure with backtracking.
Inputs: initial state s_0^d and goal state G^d.
Outputs: a feasible plan or not plan at all.

```
procedure GetPlan(s_0^d, G^d)
    repeat
        plan ← GetFastDownwardPlan(s_0^d, G^d)
        if ¬isPlanEmpty(plan) then return NULL
        end if
        action ← GetFirstAction(plan)
        success ← HasIKSolution(action)
        if ¬success then
            s_0^d ← UpdateInitialState(action)
        end if
    until success return plan
end procedure
```

It is possible that an object cannot be grasped on a certain pose, because the inverse kinematic has no solution, but it can be moved in a new pose in which it can be grasped. Therefore the pushing actions also will include the effect that the object of interest will have a solution of the inverse kinematic in the new pose. This is usually a rare case but it might happen. Of course, since the planner has no geometric information, it does not know in which pushing direction the object can be reach a position where it can be grasped, therefore this will be a totally random choice up to the planner. In the case the object is in a pose where it cannot be neither grasped or pushed because of the inverse kinematic, first the planner will return as a solution to grasp it, then it will replan and the solution will be to push it in one direction and grasp it, and so until no action can be executed and there exist no solution for the plan. In the best case, the planner will chose to push the object in direction in which the next pose will be more likely a graspable one.

A situation in which the backtracking is useful is shown in Figure 3.1. Accordingly to our strategy, the robot cannot grasp the black or red box because the gripper would collide, and the same fto push them. It has to interact with the white cup in order to

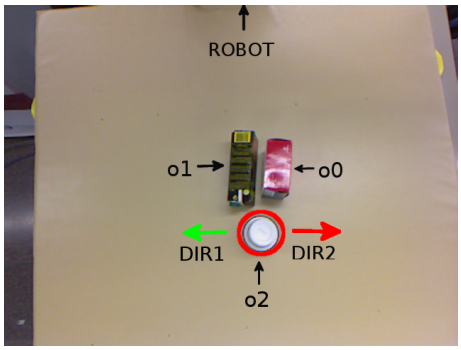


Figure 3.1: Unfeasible plan due to geometrical constraints. In this case the planner returns as action to execute grasping or pushing away the white cup (highlighted by a red circle) but it is out the configuration space of the robot and there exist no plan for that problem.

make space to move the other objects and then grasp them. The planner first returns the following plan: `(grasp o2)`, `(push_dir1 o0)`, `(grasp o1)`, `(grasp o0)`. It tries to solve the inverse kinematic for the `(grasp o2)` action, but it finds no solution. Therefore the predicate `(ik_unfeasible_grasp o2)` is added and the planner replans. The new plan is: `(push_dir1 o2)`, `(grasp o2)`, `(push_dir1 o0)`, `(grasp o1)`, `(grasp o0)`. In this case it tries to get a solution for the inverse kinematic for the pushing action but it finds no solution, so the predicate `(ik_unfeasible_dir1 o2)` is added. And so on until the planner finds that all the possible actions for a feasible plan have no solutions for the inverse kinematic. In this way the planner includes the ability to understand also why no plan exists for a certain problem.

It may happen that the object outside the working space of the robot blocks the execution of the task because it is impossible to achieve the goal to grasp all the object since one is outside the working space. This happens when all the `ik_unfeasible` predicates for an object are set to true. When this situation occurs that object is removed from the goal.

PDDL For clarity purposes, the PDDL syntax of the described actions is here reported. For the *grasping* action its PDDL syntax is shown in listing 3.1.

```
(:action grasp
    :parameters (?o - obj)
    :precondition (and
                  ; grasp it if the IK has a solution
                  (not (ik_unfeasible_grasp ?o))
                  ; grasp it if there are no objects on its top
                  (not (exists (?x - obj)(on ?x ?o)))
                  ; grasp it if there is no object that blocks it
                  ; to be grasped
                  (not (exists (?x - obj)(block_grasp ?x ?o))))
    :effect (and
            ; the object "o" is grasped
            (grasped ?o)
            ; if the object was on top of other ones now it is
            ; no more on top of them
            (forall (?x - obj)
              (when (on ?o ?x)(not (on ?o ?x))))
            ; the grasped objects no more blocks other objects
            ; to be pushed or grasped
            (forall (?x - obj)
              (and
              (when (block_grasp ?o ?x)(not (block_grasp ?o ?x)))
              (when (block_dir1 ?o ?x)(not (block_dir1 ?o ?x)))
              (when (block_dir2 ?o ?x)(not (block_dir2 ?o ?x)))
```

```
25              (when (block_dir3 ?o ?x)(not (block_dir3 ?o ?x)))
                (when (block_dir4 ?o ?x)(not (block_dir4 ?o ?x))))
```

Listing 3.1: PDDL syntax of the grasping action

For the *pushing* action its PDDL syntax is shown in listing 3.2.

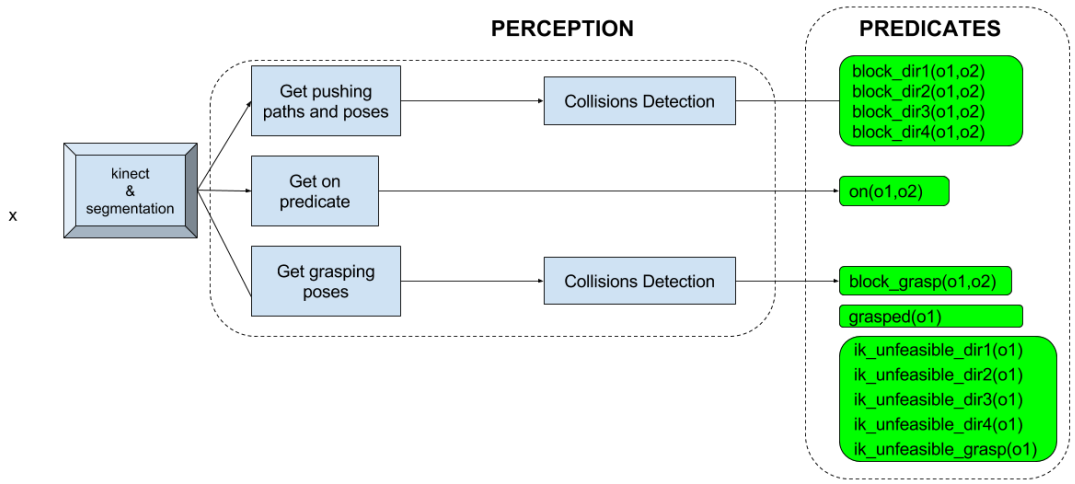
```
(:action push_dir1
    :parameters (?o - obj)
    :precondition (and
                  ; push it if the IK has a solution
                  (not (ik_unfeasible_dir1 ?o))
                  ; push in direction 1 only if there are no
                  ; objects that block it along that direction
                  (not (exists (?x - obj)(block_dir1 ?x ?o)))
                  ; push it if it has no objects on top of it
                  ; and if it not on top of other ones
                  (not (exists (?x - obj)(on ?x ?o)))
                  (not (exists (?x - obj)(on ?o ?x))))
    :effect  (forall (?x - obj)
                (and
                ; once pushed the object is no more blocked in any direction
                ; and it no more blocks other objects to be moved
                (when (block_dir1 ?o ?x)(not (block_dir1 ?o ?x)))
                (when (block_dir2 ?o ?x)(not (block_dir2 ?o ?x)))
                (when (block_dir3 ?o ?x)(not (block_dir3 ?o ?x)))
                (when (block_dir4 ?o ?x)(not (block_dir4 ?o ?x)))
                (when (block_dir1 ?x ?o)(not (block_dir1 ?x ?o)))
                (when (block_dir2 ?x ?o)(not (block_dir2 ?x ?o)))
                (when (block_dir3 ?x ?o)(not (block_dir3 ?x ?o)))
                (when (block_dir4 ?x ?o)(not (block_dir4 ?x ?o)))
                ; once pushed it can be grasped and it no more
                ; blocks other objects to be grasped
                (when (block_grasp ?x ?o)(not (block_grasp ?x ?o)))
                (when (block_grasp ?o ?x)(not (block_grasp ?o ?x)))
                ; if before it cannot be grasped because of
                ; the IK, we consider that the IK now has solution
                (when (ik_unfeasible_grasp ?o)(not (ik_unfeasible_grasp ?o)))))
    )
```

Listing 3.2: PDDL syntax of the pushing action along direction 1

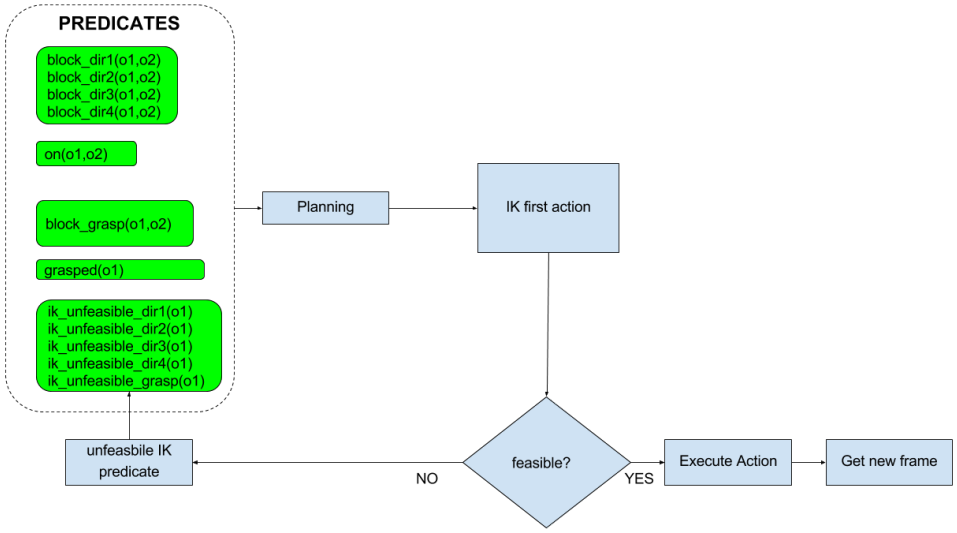
With such predicates the planner has all the basic information about the location of the objects on the scene. The symbolic predicates try to simplify the problem translating the geometric properties of the objects in symbolic ones, relaxing the planning phase. It is important to point out again that the system is deterministic, meaning that all the actions are supposed to give the resultant state with a probability of 1. Clearly the biggest uncertainty is related to the pushing action; the method used to select the pushing directions does not taken into account reliably the geometry of the object and the trajectory will be unlikely the desired one but a similar one. Overall the planner is considering to push the manipulated object at infinity to isolate it, and that's false. This is another uncertainty of the pushing action due to the lack of geometrical information.

After the execution of an action the planner gets a new depth image from the Kinect, it segments the scene, it recomputes the predicates and replans. In this way the planner considers a totally new problem and all the uncertainties associated to the previous plan will be solved by the current one.

In order to provide a graphical scheme to the reader the perception and planning pipeline is depicted in Figure 3.2. It can be appreciated the two fold strategy of the algorithm, the



(a) Perception Pipeline



(b) Planning Pipeline

Figure 3.2: Perception and planning pipeline

first stage (Figure 3.2a) is devoted to get the predicates from the Kinect sensor and the second one to evaluate the feasibility of the returned plan and toe xecute it(Figure 3.2b).

How the predicates are computed will be discussed in detail in the next chapter.

4. Implementation

In this chapter the implementation is discussed, presenting how the objects are detected and how the symbolic predicates are obtained.

4.1. Object Localization

To interact with the objects we have first to detect them. More correctly, they need to be segmented since the algorithm is dealing with unknown objects it is not going to recognize an object as a particular one, instead it segments the objects in the scene.

This stage is composed of 3 steps:

1. Filtering the point cloud
2. Detecting the tabletop objects
3. Segmenting the tabletop objects

The Kinect camera is recording a depth image which is quite noisy, overall at the edges, for a proper segmentation is better applying a filter. Fort this we used the statistical outlier removal algorithm [37]. Next, since the objects are on a table, the algorithm has to detect first the table, and so the objects that stand on top of it, and then segmenting them. We don't want to segment the entire image, if so, the table would be segmented as an object, and the floor as well.

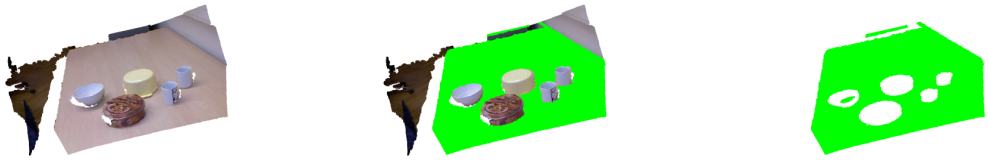
4.1.1. Tabletop Object Detection

The strategy for the tabletop object detection phase is composed of 3 different steps:

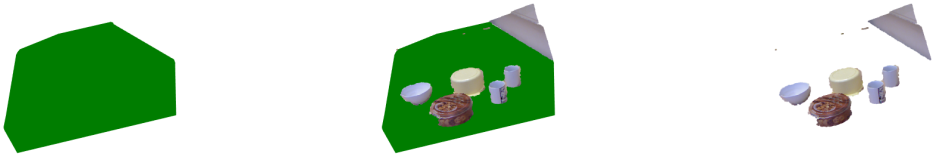
1. **Table plane estimation** (by RANSAC): the points of the table are detected estimating a plane in the point cloud, all the points which belong to such a plane are the points of the table.
2. **2D Convex Hull of the table**: having the points of the table a 2D convex hull is computed in order to get a 2D shape containing those points.
3. **Polygonal prism projection**: all the points are projected on the table plane previously estimated and all the points which projections belong to the 2D convex hull are considered to be points of tabletop objects. The points that do not belong to it are points of non-tabletop objects.

The steps of this tabletop object detection algorithm are described in Figure 4.1 for the point cloud[1] in Figure 4.1a.

[1]Point cloud taken from the Object Segmentation Database (OSD) http://users.acin.tuwien.ac.at/arichtsfeld/?site=4



(a) Input Point Cloud **(b)** RANSAC plane estimation **(c)** Table plane



(d) Convex Hull **(e)** Convex hull in the point cloud **(f)** Tabletop objects

Figure 4.1: Object Segmentation: Given the point cloud (a), the estimated table's plane is obtained (b and c), its convex hull is extracted (d and e), and the tabletop objects are obtained by a polygonal prism projection (f).

4.1.2. Object Segmentation

Once the objects on the table are detected the following phase is to segment them in order to get a point cloud per object.

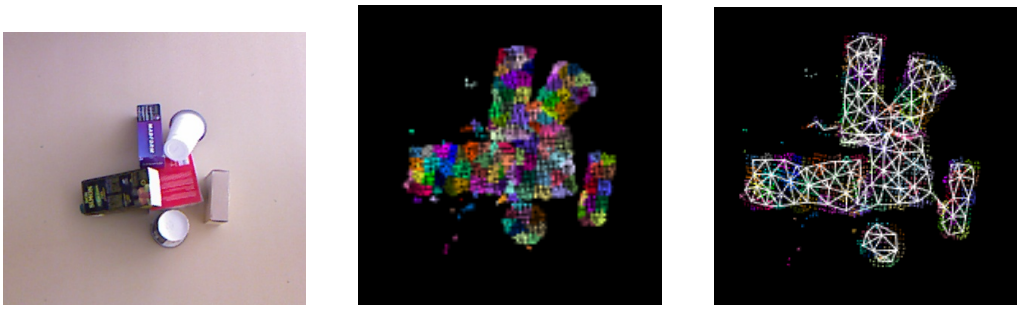
Supervoxel

For their segmentation the supervoxel concept is used. A supervoxel is a group of voxels that share similar characteristics, for instance similar normals.

In this work the supervoxels are computed with the *Voxel Cloud Connectivy Segmentation* (`VCCS`) algorithm [33], which is able to be used in online applications. An example of the obtained supervoxels is shown in Figure 4.2.

The algorithm works in 3 main steps:

- Voxelizing the point cloud
- Creating an adjacency graph for the voxel-cloud
- Clustering together all adjacent voxels which share similar features



(a) Input scene **(b)** Supervoxels **(c)** Adjacency graph

Figure 4.2: Example of supervoxels for the table top objects.

Local Convex Connected Patches Segmentation

Once the supervoxels of the tabletop objects are computed, they can be clustered in order to segment the objects. Papon et al. also proposed a segmentation algorithm based on their supervoxel technique, called *Local Convex Connected Patches Segmentation* (`LCCP`) [38]. This algorithm permits to segment objects by clustering together adjacent convex supervoxels, in Figure 4.3 the algorithm is briefly described. The algorithm is quite simple but very good for segmentation of objects that have convex shapes.

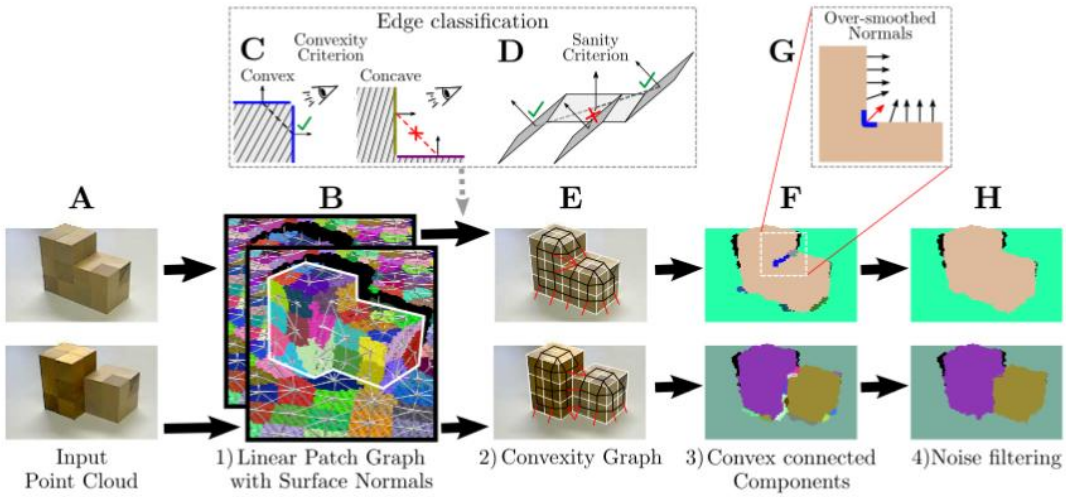


Figure 4.3: `LCCP` algorithm's structure. Reproduced from [38]

It clusters all the adjacent convex supervoxels (patches) using 2 criterion:

- Extended criterion: to consider two adjacent patches convex, both must have a connection to a patch which is convex with respect both patches
- Sanity Criterion: check if the adjacent patches which can be considered as convex present geometric discontinuities (see point D of Figure 4.3), in this case they are not considered as valid to form a cluster.

Then, due to the smoothed normals that could appear in some edges of the objects (point G Figure 4.3), the algorithm merges the clusters that are composed of few supervoxels to the biggest adjacent cluster.

By tuning properly the parameters of the segmentation algorithm the objects can be correctly segmented obtaining for one of them a point cloud. Two examples of the segmentation algorithm for a cluttered scene are depicted in Figure 4.4.

Figure 4.4: Example of segmentation results.

Note that we set the algorithm in order to segment considering geometric properties, and not the color of the pixels. Considering the colors could lead to worst segmentation results for our case of studio since many objects have several colors.

A color based segmentation could segment a draw on an object, or a small part of the object, as a different object, but this, accordingly to the strategy we are going to use (see next sections), would lead to an unfeasible problem. For instance, in Figure 4.5 is shown a box with a green stripe on its top surface, a segmentation algorithm based also on colors could lead to segment the green stripe as another object, and the result is that it is impossible to grasp the green stripe without collide with the box. This is the main reason the segmentation we used is based only on geometric features.

Figure 4.5: Box with a green stripe.

4.2. Background

In this section some concepts, that will be used to execute the actions and to generate the states, are presented.

Principal Direction The principal direction of an object is its principal axis which is defined as any of three mutually perpendicular axes about which the moment of inertia of a body is maximum. For instance, for a rectangular object its principal direction is the axis aligned with its longest dimension.

To obtain the principal axis the principal component analysis (PCA) [18] technique is used. This technique is a common statistical procedure that uses orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, which are called principal components. The transformation is defined in a manner that the first component has the largest variance, the second has the second largest variance and so on. The principal components are orthogonal because they are the eigenvectors of the covariance matrix, which is symmetric. An example of the principal components for a 2D data set is depicted in Figure 4.6a[2]. The principal components are computed through the covariance matrix of the set of observation, and its eigenvectors $\bar{\lambda_v}$ represent the principal components while its eigenvalues λ represent the variance of the data set along the principal component $\bar{\lambda_v}$.

A generic point cloud can be seen as a set of observations and the PCA can be directly applied to the object's point cloud to retrieve its principal components. In this works we refers to principal components as principal directions. In Figure 4.6b the first two principal directions of a generic object are illustrated. Note that for each principal direction we can actually obtain two directions (one per sense).

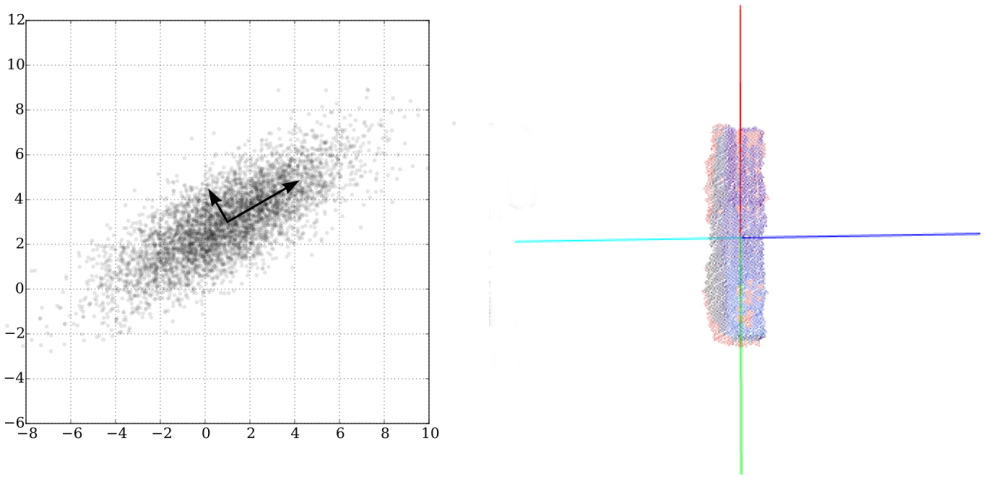
Projection onto a plane We will see later that the concept of the projections of a point into a plane will be useful. Considering a point $p = (x_p, y_p, z_p)$ and a plane $\mathcal{P}$ defined by the following equation

$$ax + by + cz + d = 0 \tag{4.1}$$

the projection $p_{\mathcal{P}}$ of point p onto the plane $\mathcal{P}$ is given by the following set of operations:

1. Given the origin point $\mathcal{P}_O = (x_O, y_O, z_O)$ of the plane, which can be calculated by

[2]Image taken from https://en.wikipedia.org/wiki/Principal_component_analysis



(a) **(b)**

Figure 4.6: Principal Components Analysis - In Figure 4.6a PCA for a standard 2D set of observations. In Figure 4.6b results of the PCA for a rectangular segmented object. The green, red lines refers to different ways of the first principal direction, while blue and cyan lines refers to different ways of the second principal direction. The third one is orthogonal to the first two principal directions.

arbitrary x_O and y_O coordinates as

$$z_O = \frac{-1}{c}(ax_O + by_O + d),$$

calculate the coordinates of $\mathcal{P}_O$ with respect the point p

$$^{p}\mathcal{P} = p - \mathcal{P}_O,$$

2. then calculate the projection of $^{p}\mathcal{P}$ onto the plane normal $\bar{n} = (a, b, c)$

$$\lambda_p = \bar{n} \cdot {}^{p}\mathcal{P},$$

3. translate point p by λ_p along the normal of the plane $\bar{n}$

$$p_P = p - \lambda_p \bar{n}.$$

The minus sign is due to the fact that the normal is pointing upward.

Rotation Matrices Rotation matrices are matrices which express a rotation between two reference frames. Given two frames $\{A\}$ and $\{B\}$, and the rotation matrix ${}^{A}_{B}R$ that defines the rotation of $\{B\}$ relative to $\{A\}$ then a point ${}^{A}P$ with respect frame $\{A\}$ is given by ${}^{A}P = {}^{A}_{B}R\,{}^{B}P$, where ${}^{B}P$ is the same point relative to frame $\{B\}$.

Having a frame $\{B\}$ defined by axis ${}^{A}\hat{X}_B$, ${}^{A}\hat{Y}_B$ and ${}^{A}\hat{Z}_B$, where ${}^{A}\hat{Y}_B$ is the y axis of frame $\{B\}$ relative to frame $\{A\}$, the rotation matrix between $\{A\}$ and $\{B\}$ is defined as

$${}^{B}_{A}R = \begin{bmatrix} {}^{A}\hat{X}_B \\ {}^{A}\hat{Y}_B \\ {}^{A}\hat{Z}_B \end{bmatrix}$$

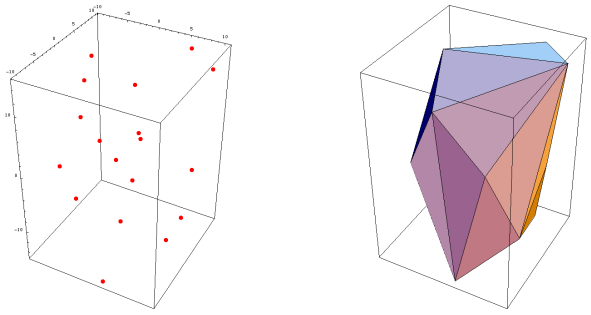
To transform any object, such as the gripper mesh model, to a pose defined by frame $\{B\}$ then the following homogeneous transform is applied:

$$H = \begin{bmatrix} {}^{A}_{B}R & {}^{A}B_{O} \\ \bar{0} & 1 \end{bmatrix}$$

where ${}^{A}_{B}R = {}^{A}_{B}R^{\top}$ and ${}^{A}B_{O}$ is the origin of frame $\{B\}$ relative to $\{A\}$. In this way, having some axis that define our new reference frame, we can transform the gripper model in such a way its closing point is in the origin of the new frame and its orientation is aligned to the one of the new reference frame.

Bounding Box A bounding box is the smallest cubic volume that completely contains an object[3]. An axis-aligned bounding box (AABB) is a bounding box aligned with the axis of the coordinate system, while an oriented bounding box (OBB) is a bounding box oriented with the object. To compute the OBB the object is transformed from its frame to the world frame and the dimensions of the bounding box are obtained by computing the maximum and minimum coordinates of the transformed object. In this way it is possible to have an approximation of the length, width and height of an object.

Convex Hull A convex hull of a point cloud P is the smallest 3D convex set that contains P. In Figure 4.7[4] an example of the convex hull for a point cloud is shown. The vertices are



(a) Pointcloud **(b)** Convex hull

Figure 4.7: Convex hull example

first detected and then connected among them by means of triangles. In this way a triangle mesh is associated to the convex hull.

Collision Detection To understand if an object blocks a certain action, such as the pushing along a direction, we have to check if along the desired trajectory the pushed object will collide with the other ones. The collision detection is therefore a crucial step to generate the states. There exist different techniques to assert if two objects are colliding and all of them need a representation of the object, which could be a basic shape or a more complex as an octree.

The mesh shape has been thought to use since it can be directly obtained from a convex hull.

Given two objects A and B and their configurations $\mathbf{q_A}$ and $\mathbf{q_B}$, the collision test returns a boolean value about whether two objects collide or not [32]. Two objects collide if

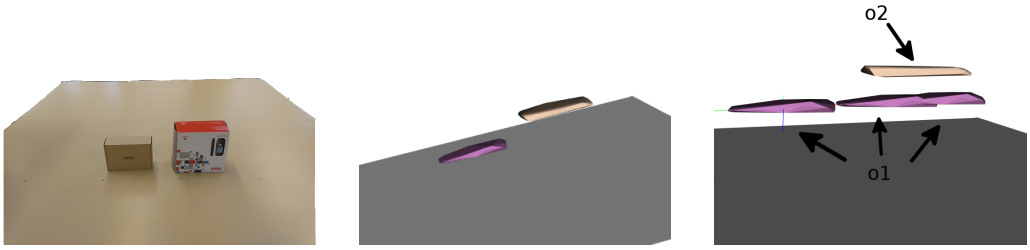
$$A(\mathbf{q_A}) \cap \mathbf{B}(\mathbf{q_B}) \neq \mathbf{0}$$

[3] `https://en.wikipedia.org/wiki/Bounding_volume`
[4] Images obtained from `http://xlr8r.info/mPower/gallery.html`

The collision detection will be used to understand if in a given pose $\mathbf{q}$ the object A would collide with the other objects in the scene.

In order to relax the collision detection majority of collision libraries, before to use complex algorithm to detect collision between two shapes, they first check if the bounding volumes (e.g. AABB) of the objects intersect, if they don't the objects surely don't collide. If their bounding volumes intersect the objects might collide.

Objects Modelling The Kinect, given its pose, is only able to see mainly the top surface of the objects and not all the sides, and therefore we cannot apply directly the convex hull algorithm to the detected surfaces. If we would apply the convex hull on an object's surfaces, we will have likely the situation depicted in Figure 4.8c, in which the collision detections would not detect any collision when it should. This is because we are totally missing the rest of information about the geometry of the object since we only know the objects top surfaces.



(a) **(b)** **(c)**

Figure 4.8: Convex hulls and collision detection using the segmented objects retrieved by the `LCCP` segmentation algorithm. The gray surface represents the plane's 2D convex hull. In Figure 4.8b it is possible appreciating that we miss the information about the hidden part of the object. In Figure 4.8c a collision detection example is depicted. The convex hull of object `o1` is translated along a direction and no collision is detected since the two convex hulls do not intersect.

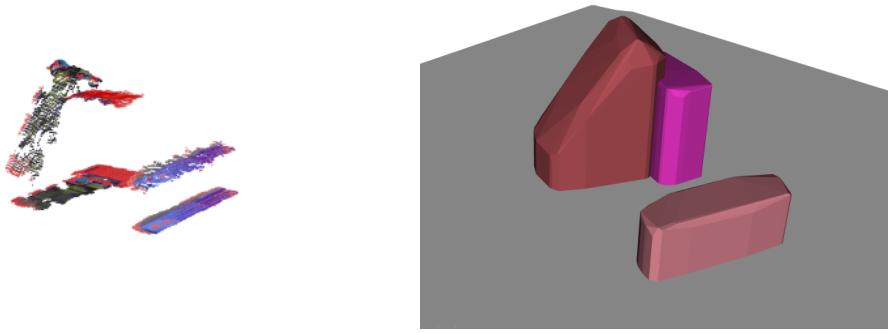
From the Kinect's point cloud also the table plane is known, so the information we have are: the table plane model and the segmented objects (mainly the top surfaces). If an human would be in the same pose of the Kinect, looking at the table, it will imagine that the objects are not floating surfaces, and he/she will deduce the objects shape from the shape of the top surface. The sides of the objects can be deduced by projecting the top surface's edges to the plane and then filling the missing object's sides with points. To do that we have to detect the top surface's edges. A easier method is directly projecting all the points of the surfaces onto the table plane and then apply the convex hull algorithm to the resulting point cloud given by the sum of the top surface and its projection. In this way the missing sides are indirectly retrieved by the convex hull. An example of this method is depicted in Figure 4.9.

4.3. Actions Execution

Since the generation of the state depends on the way we decided to execute the actions, the way the actions are executed is discussed in this section.

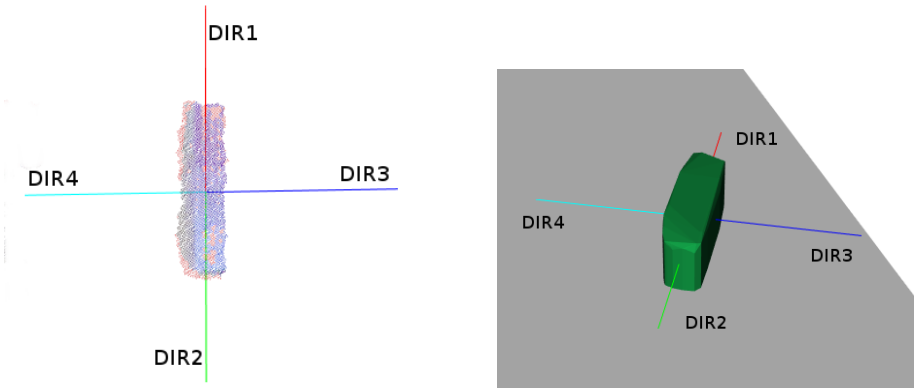
4.3.1. Pushing

Pushing is a difficult action to execute when the goal is to move one object along a path. The majority of pushing actions in object manipulation have the aim to interact with the objects in order to move them and validate the segmentation [21] [19] [20] , without taking



(a) Surfaces projection **(b)** Resulting convex hull

Figure 4.9: Convex hull of the objects using their projections onto the table plane.



(a) Pushing directions computed using the segmented surface seen by the Kinect.

(b) Pushing directions associated to the object's convex hull.

Figure 4.10: Example of pushing directions.

care about the final position of the objects or about eventual collisions. Hermans er al. [17] presented a novel algorithm for object singulation through pushing actions, here the pushing actions have the aim to separate objects in cluttered scenes but they are not interested in singulate tidily the objects but just to find a feasible interaction to singulate them regardless possible collisions.

When we push an object all the object's geometry has to be taken into account in order to find a suitable pushing direction. Moreover, humans push objects with a certain initial direction chosen accordingly the object geometry and then adapt the pushing direction accordingly to the error with respect the desired one. This would imply the implementation of a controller which is out the scope of this thesis.

We considered to work with objects with simple shapes, such as parallelepipeds. A human would push such object mainly accordingly its principal axis and the one orthogonal to its (i.e. the first 2 principal components of Figure 4.6b), he/she also could push it along the diagonal thanks to the several degrees of freedom of the hands. Inspired by this consideration, we decided to consider as possible directions to push an object its first two principal directions. In particular, there are two senses for each direction, so in total we have 4 possible pushing directions per object, as depicted in Figure 4.10.

Another things to take into account is that the principal directions are not always parallel to the table plane. An object which stands on top of a table will be obviously pushed along a direction parallel to the table. As previously described in Section 4.2 a point

can be projected on to the table plane, but a point and a vector (the principal directions are vectors) have the same representation, and therefore we can apply exactly the same equations to obtain the projections of the principal components onto the table plane. So the pushing directions considered are nthe principal directions but their projections.

Next, having some pushing directions along with the robot will push the object, the pose for the the gripper to push that object has to be decided. The pose has been chosen accordingly to the shape of the gripper.

In Figure 4.11a is possible observing the profile of the gripper mounted to the base of the gripper, highlighted by the blue color. Such base has a circular shape and the gripper's deep is less then the one of the base. It is undesirable pushing an object with a circular, or spherical, shape for the end effector because there is more uncertainty on the resulting path of the manipulated object. The gripper has no a circular shape and it is all symmetric, this make it suitable to push an object with a certain stability (i.e. make the object follow the desired path) during the action. Since we want a pushing action accurate as most as possible, we don't want that the gripper's base touches the manipulated object. Knowing

(a) Gripper's profile view gripper's base.

(b) Profile view of a desired pose for pushing an object.

Figure 4.11

also the height of the objects retrieved by its OBB, it is possible having a pose for the gripper is such a way that the gripper's base does not touch the object. The gripper's pose, relative the object, is computed in manner to locate the red point of Figure 4.11a to be at the same height of the object. In this way the fingers will fully touch the object during the pushing action. Moreover, to make easy for the robot reaching the pushing pose, it was defined to be a certain distance from the object (in our experiment it was set to 5cm). It would be difficult to reach a pose of Figure 4.11b without colliding with the object.

Due to the limited opening width of the gripper (8 centimetres) the object the robot is going to manipulate have small width. This means that when pushing along the principal axis, the object's width is likely small. Such a situation is depicted in Figure 4.12a. Pushing in such a way the gripper will likely push also the black juice box. Therefore when pushing along the principal axis the pose is chosen to be the one in Figure 4.12b. Of course is more stable a pushing pose like the one in Figure 4.12a, since the contacts point (the fingers) are more distant. For this reason that pose is used only when pushing the objects along a direction 3 and 4.

Having the projections of the principal components, the table normal and the desired coordinates for the gripper's closing point it is possible defining a rotation and translation matrices. To push along direction 1 those matrices are:

(a) **(b)**

Figure 4.12: Possible pushing poses for push an object along its principal axis - In Figure 4.12a the closing direction of the gripper is orthogonal the pushing direction, and for the case depicted in the figure the gripper will likely push also the black juice box. In Figure 4.12b the closing direction of the gripper is parallel to the pushing direction.

$$R_{dir1} = \begin{bmatrix} dir2_X & dir2_Y & dir2_Z \\ dir4_X & dir4_Y & dir4_Z \\ n_x & n_y & n_z \end{bmatrix}^\top \qquad T = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \tag{4.2}$$

where $dir1_X$ refers to the x coordinate of the vector that defines the direction 1, n is the table normal and c is the desired tool central point.

As previously said, the planner consider to push the objects at infinity, in the reality the robot has to push the objects for a finite length. Since the planner has no geometric information, the pushing length is chosen accordingly the dimension of the AABB relative to the pushing direction. For instance, if the robot is going to push the object `o1` along direction 1 the length l of the pushing action is $l = k \cdot AABB_{\texttt{o1}}(length)$, where k is a gain factor (1 in our experiments) called *pushing step*. This is a big limitation since the robot will push an object for a length which is function of the manipulated object and not to surrounding ones.

To retrieve the path we consider the total length l and we discretize it by n points having in this way $n+2$ poses (+2 because of the pushing pose and the final pose). For each pose the inverse kinematic is done. In this way we obtain a discrete path.

When the robot approaches the pushing pose it could be that it collides with other objects. It would be suitable to use *MoveIt!*[5] which can build an octomap representation of the scene and find a path avoiding collisions with the scene. The integration of *MoveIt!* will be a future work. To avoid the collisions we considered a pre-pushing pose which has the same pose of the pushing pose but translated, accordingly to the table's normal, 10 centimetres from the pushing pose. This pre-grasping pose is easier to reach without collisions. After the execution of the pushing action the robot goes to its *home* pose (depicted in Figure 1.5a) in order not to stay inside the Kinect's view. When it goes to home it might happen that it collides with some objects, so also for the final pose with consider another one translated, accordingly the table normal, 10 centimetres from the last pose. In this way the pushing trajectory is defined by a total of $n+4$ poses.

[5]Ioan A. Sucan and Sachin Chitta, "MoveIt!", [Online] Available:`http://moveit.ros.org`

4.3.2. Grasping

There exist an advanced state of the art regarding grasping. Despite this all the techniques of grasping are very computationally expensive. Many of them rely on the identification of the shape of the objects and then a set of pre-built grasping poses is returned[6]. Other techniques rely on the identification of local features which can state if a grasping pose is feasible or not. Two goods grasping planning algorithms of this kind, which deal with novel objects, are AGILE [39] and HAF [15], despite this, they are not so robust and they are computationally expensive and not suitable for this thesis[9]. In order to have a fast planning algorithm we considered a very simple approach to grasp the objects, which is suitable only with the kind of objects we are going to interact with. Despite this, the planner presented by this thesis can be directly integrated with several grasping algorithms.

The idea is to grasp the object in manner that the gripper's closing direction[6] is orthogonal the principal axis of the object. The approaching direction[7] of the gripper is given by the third principal component of the object. Then the gripper's closing point coordinates are given by the centroid, of the object's top surface, translated along the approaching direction by the half of the gripper's fingers height. In this manner a single grasping pose is obtained for each object.

To grasp the object also the robot needs a pre grasping pose, if not the gripper would collide with object attempting to reach the grasping pose, moving it away, and the grasp would fail. The pre grasping pose is simply defined by the grasping pose translated along its approaching direction by 10 centimetres.

On the whole the grasping action is composed of the following set of actions:

1. Reaching the pre grasping pose
2. Opening gripper
3. Reaching grasping pose
4. Closing gripper
5. Reaching pre grasping pose again: this is done in order to avoid collisions between the grasped object and the other ones
6. Going to the dropping pose: the object will be dropped into a bin

4.4. States generation

In Chapter 3 the predicates used are described, in this section their computation is presented in detail.

4.4.1. Predicate: `block_grasp`

The `(block_grasp o1 o0)` predicate refers to the fact that object `o1` blocks `o0` to be grasped. The computation is this predicate is straightforward: the mesh model of the opened gripper is transformed to the grasping pose of object `o0`, and checked if it collides with the other objects. In figure 4.13 such procedure is shown and in Algorithm 2 the pseudo algorithm is described in detail.

[6]The gripper's closing direction is the direction along with the fingers move when grasping.
[7]The gripper's approaching direction is the direction along with the gripper approaches the grasping pose.

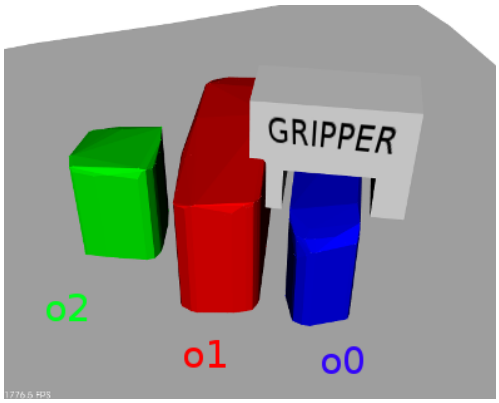


Figure 4.13: Visualization of the computation of `block_grasp` predicate for object `o0`. The opened gripped model is transformed to the grasping pose for object `o0` and it is tested if the gripper mesh model collides with the other objects, in this case it collides with `o1`.

Algorithm 2 Computation of `block_grasp` predicates.
Inputs: Set of objects O (convex hull retrieved with the projection onto the table plane) and the set of grasping poses G_{poses}.
Outputs: The `block_grasp` predicates.

```
function ComputeBlockGraspPredicates(O,G_poses)
block_grasp_predicates ← NULL
    for all A ∈ O do
        gripperMeshTransformed ← trasnformGripperModel(G_poses(A))
        for all B ∈ O do
            if A ≠ B then
                collision ← isThereCollision(gripperMeshTransformed,B)
                if collision then
                    block_grasp_predicates ← AddPredicate((block_grasp B A))
                end if
            end if
        end for
    end for
return block_grasp_predicates
end function
```

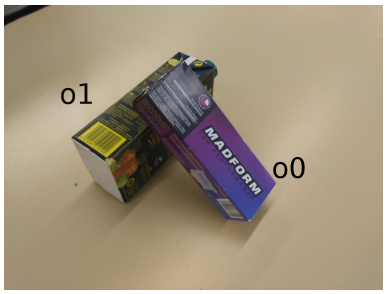
Notice that this method implies to check for collision between the gripper and objects that might be very far from the interested object, i.e. there is no need to compute the collision detection. Despite this, as explained in Section 4.2, the majority of collision detection algorithms first check if the bounding boxes of the objects intersect. This is a computationally cheap operation, and only if they intersect the computationally expensive algorithm are used to check for collision. This makes the Algorithm 2 efficient and computationally not expensive. It has been observed that, in average, to compute this predicate the time is about 10 milliseconds per object.

4.4.2. Predicate: `on`

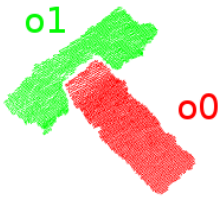
The `(on o0 o1)` predicate means that object `o0` is on top of object `o1`. With the convex hull of the objects is easy to understand if two objects are one on top of the other one by checking for collision, but in this way we do not know who is above and who is below. To do this their surface projections onto the table plane are used. The research group of

Artificial Intelligence and Robotics Laboratory of Istanbul Technical University, published some interesting researches suitable to the aim of this thesis. In [13] [31] [14] the authors proposed some approaches to enhance 3D recognition and segmentation results to create and maintain a consistent world model involving attributes of the objects and spatial relations among them. Their researched focused on modelling the world for manipulation planning tasks. They do not consider scene like the one of this thesis but simpler ones such as a pile of cubes above each other. What can be directly used from their work is the computation of the `on` predicate. The *on* relation for a pair of objects is determined by checking whether their projections onto the table plane overlap. This predicate was not a relevant part of their work and they did not provide too much information about its computation. Therefore our implementation for the `on` predicate is based on their idea with some modifications.

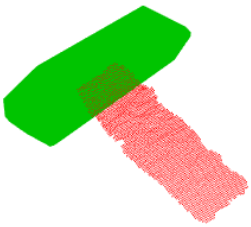
Our idea is based on the fact that an object which stands on top of another one occludes some parts of the object below. In the other side, the one below does no occlude any part of the top object. Let's consider the scene in Figure 4.14a, the object `o0` occludes a portion of object `o1`. The projections P_0 and P_1 onto the table plane of `o0` and `o1` are respectively the red and green ones in Figure 4.14b. This means that the convex hull C_{P_1} of the projection P_1 of `o1` intersects with the projection P_0 of `o0`, while the projection P_1 of `o1` does not interesct with the convex hull C_{P_0} of the projection of `o0` (Figures 4.14c and 4.14d).



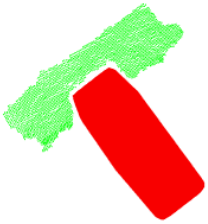
(a) Scene



(b) P_0 & P_1



(c) C_{P_1} & P_0



(d) C_{P_0} & P_1

Figure 4.14: Visualization of the computation of the `on` predicate. Figure 4.14b shows the real image, Figure 4.14b shows the projections of the objects onto the table plan while Figures 4.14c and 4.14d represent the two steps strategy to compute the `on` predicate.

Although this method works fine to compute the `on` predicate it has the limitation that its scope is only for objects for a rectangular shape.

It is important to take into account also that actually the edges of the occluded parts of the below object, once projected, could be at the same position, of some projected edges of the top object. This could be dangerous for the computation of this predicate. Therefore a threshold is added. Focusing the attention on Figure 4.14c it can be appreciated that the

intersection $C_{P_1} \cap P_0$ includes several points, while, in case the edges projections relative to the occluded and occluding part have similar coordinates, the intersection $C_{P_0} \cap P_1$ would include just few points. Therefore the `(on o0 o1)` predicate is update accordingly to the following formula:

$$\texttt{(on o0 o1)} = \begin{cases} True, & length(C_{P_0} \cap P_1) < th_0 \ \ \wedge \ \ length(C_{P_1} \cap P_0) > th_1 \\ False, & otherwise \end{cases} \tag{4.3}$$

where $length(A)$ means the number of elements in the set A. The values of the thresholds th_0 and th_1 are determined empirically and they are $th_0 = th_1 = 100$.

The formula 4.3 is then evaluated for every possible combinations of objects, that is $n(n-1)$ times, where n is the number of objects. Despite this, its computation is very fast. The example in Figure 4.14 was evaluated just 2 times since there are only 2 objects and it took 3 milliseconds, that is $\approx 1.5 \frac{ms}{pair\ of\ object}$. For instance, for a complex scene with 10 objects the total time devoted to compute this predicate would be approximatively $10 \cdot 9 \cdot 1.5 \approx 135ms$.

4.4.3. Predicate: `block_dir`$_i$

The `(block_dir`$_i$ `o1 o0)` predicate, if true, means that object `o1` blocks object `o0` when moved along its i-th direction.

Object `o1` can blocks object `o0` to be moved along a certain direction if a collision will appear between the two objects. In order to do that, having a certain pushing length l_i for the i-th direction of object `o0`, its convex hull $C_{\texttt{o0}}$ is translated along the considered direction until reaching its final position which is $p_f = p_i + l \cdot dir_i$, where p_f and p_i are respectively the centroid at the final and initial pose. Object `o0` is going to do a path from its initial and final pose so the collision should be checked along its path. We decided to use a discrete strategy, that is we consider several poses between the initial one and the final one, including the final one, and for each one we check if the transformed object collides with the other objects.

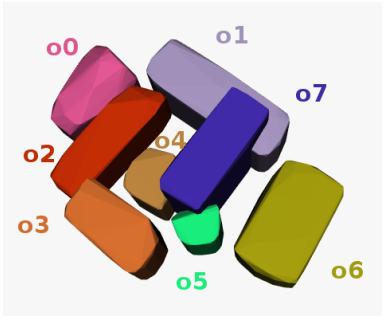
Considering the kind of objects we are going to interact with, the discrete path is computed by translating the object along the pushing direction by a length equal to the OBB dimension associated to that direction, until reaching the final pose. Note that no collision detection is done for the object at its initial pose, therefore the total number of poses considered for the pushing path are $n_{poses} = \left\lceil \frac{k \cdot AABB_{dimension}}{AABB_{dimension}} \right\rceil = \lceil k \rceil$, where k is the pushing step defined in Section 4.3.1.

To push an object along a certain direction the robot needs to put its end effector at the opposite side of the object. Therefore object `o1` can block object o0 to be moved along a certain direction also in the case the end effector cannot be put in the pushing pose because it would collide with `o1`. This computation is simply done by transforming the closed gripper mesh model to the pushing pose and check for collision with the other objects, as similarly done for the computation of the `block_grasp` predicate.

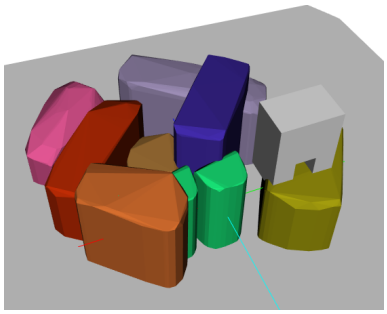
In Figure 4.15 is shown graphically the procedure to compute the predicate.

Note that also the gripper during the pushing action will move, so ideally the collision checking should be done exactly as done for the object. We decided to neglect this and check only for the initial pushing pose in order to relax the planner and not to make it too much conservative. This means that during the pushing action the robot might actually move more than one object. Despite this relaxing strategy the algorithm showed to work perfectly and cases in which the gripper moved more than one object were really rare.

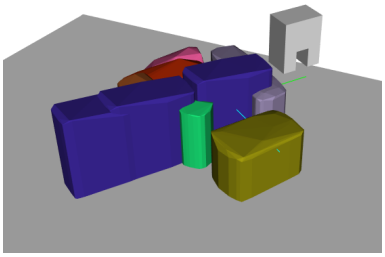
The computation of this predicate can be appreciated in detail in Algorithm 3.



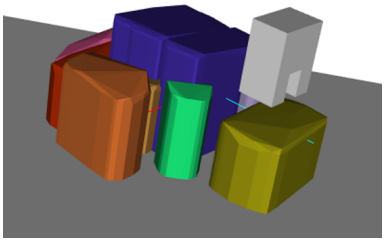
(a) Top view of the convex hull of the segmented objects.



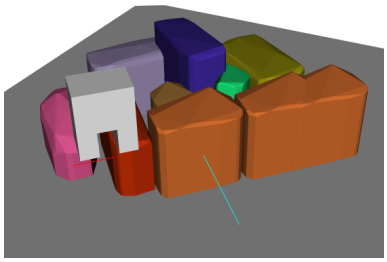
(b) Evaluating the `(block_dir1 * o5)` predicate



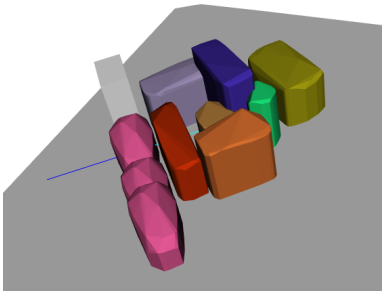
(c) Evaluating the `(block_dir1 * o7)` predicate



(d) Evaluating the `(block_dir3 * o7)` predicate



(e) Evaluating the `(block_dir2 * o3)` predicate



(f) Evaluating the `(block_dir1 * o0)` predicate

Figure 4.15: Visualization of the computation of $\texttt{block\_dir}_i$ predicates. "Evualuating the `(block_dir1 * o0)`" means that the algorithm is evaluating for all the objects, except `o0`, if they collide with `o0` when pushed along direction 1.

The computation of this algorithm is the most computational expensive of all the planner since it involves many collision detection phases. In fact the time required to compute this predicate for the example in Figure 4.15 (considering to push the objects 1.5 times the OBB dimension relative to the pushing direction) was $\approx 1.115 seconds$, were $\approx 0.869s$ were dedicated to check if the objects would collide when moved, while $\approx 0.239s$ were dedicated to check if the gripper collides with some objects.

Algorithm 3 Computation of `block_dir` predicates.
Inputs: set of objects O (convex hull retrieved with the projection onto the table plane), set of the pushing directions P_d of all the objects, set of the initial pushing pose P_{poses} of all the objects, set of all the pushing paths P_{path} relative to each direction and each object.
Outputs: `block_dir` predicates

function COMPUTEBLOCKDIRPREDICATES(O,P_d,P_{poses},P_{path})
`block_dir_predicates` ← NULL
  for all $A \in O$ **do**
    for all $d \in P_d(A)$ **do**
      for all $p \in P_{path}(A, d)$ **do**
        $A_T \leftarrow$ TRANSFORMOBJECT(A, p)
        for all $B \in O$ **do**
          if $A \neq B$ **then**
            collision ← ISTHERECOLLISION(A_T,B)
            if collision **then**
              `block_dir_predicates` ← ADDPREDICATE(`(block_dir`$_d$ `B A)`)
            end if
          end if
        end for
      end for
      closedGripperMesh ← TRASNFORMCLOSEDGRIPPERMODEL($P_{poses}(A, d)$)
      for all $B \in O$ **do**
        if $A \neq B$ **then**
          collision ← ISTHERECOLLISION(*closedGripperMesh*,B)
          if collision **then**
            `block_dir_predicates` ← ADDPREDICATE(`(block_dir`$_d$ `B A)`)
          end if
        end if
      end for
    end for
  end for
return `block_dir_predicates`
end function

5. Software design

In this chapter the design of the software is briefly described and the external libraries used as well.

The code has been implemented in the ROS framework (Robot Operating System) [34] using `C++` as programming language. The external libraries used are Point Cloud Library (PCL) [36], an open source library which provides a wide variety of tools for 3D perception, and the Flexible Collision Library (FCL) [32], an open source library for collision checking.

The algorithm is mainly based on the PCL library which is used to do the following operations: filtering, segmentation, plane estimation, principal component analysis, projections onto the table plane and convex hulls. The FCL library was used only for collision detection between the convex hulls of the objects and the gripper as well.

The planner used is the Fast Downward planner [16]. To get a plan the binary file of planner is called giving as inputs a domain and a problem description in PDDL syntax.

ROS The algorithm has been developed by using different nodes in order to have a modular code. The nodes implemented are:

- a node to segment the objects and estimating the table plane coefficients,
- a node to generate the states having as input the segmented objects and the table plane,
- a node that, given the states, call the Fast Downward binary file and returns the plan,
- a node to execute the first action of the plan,
- a decision maker node which control all the process and decides the next task to do.

These nodes are implemented as service, that is they receive an input and return an output. The software architecture is sketched in Figure 5.1. Once the decision maker receives a point cloud from the Kinect it calls the `segmentation.srv` service giving as input the point cloud and it receives as results a point cloud per tabletop object and the plane coefficients. Then it calls the `states.srv` service giving as input the table top objects point cloud and the plane coefficients and it gets as output all the states, as well the poses for the pushing and grasping actions. Then it sends the states to the task planner by requesting the `plan.srv` service. Finally it executes the first plan's action calling the `execute.srv` service. The result of this service is a boolean variable which specifies if the requested action has the inverse kinematic feasible. If it is not feasible, it add to the states the `ik_unfeasible` state for that action and recall the `plan.srv` service until a feasible plan is returned or there exist no plans. Once the action has been executed, it waits to receive a point cloud from the Kinect and all the process is repeated until no objects stand on the table or no feasible plan exists.

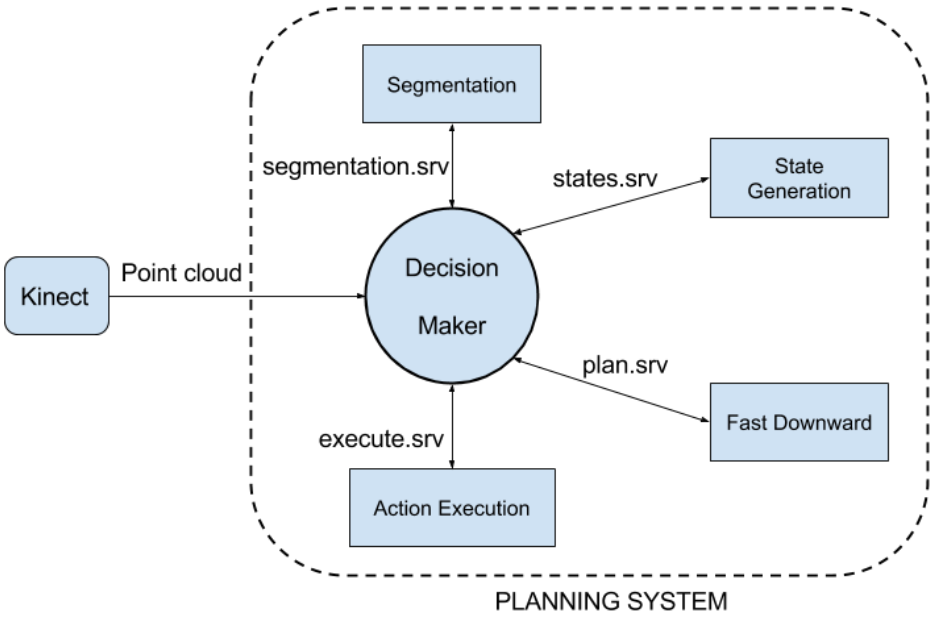
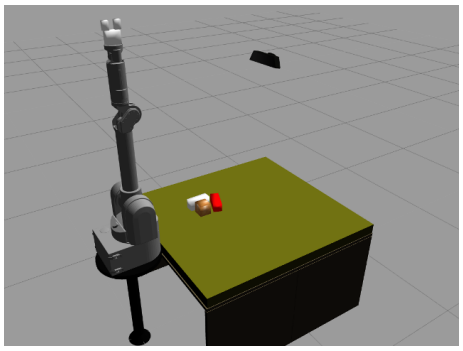


Figure 5.1: Software architecture.

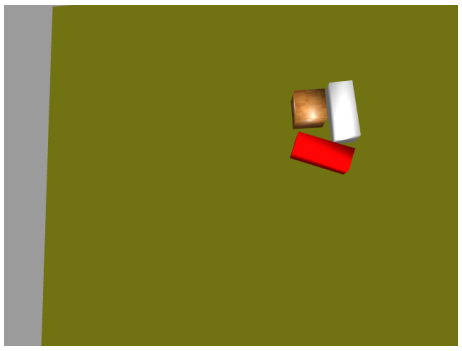
Simulation Before to test the implemented algorithm with the real robot it was first tested in simulation with Gazebo[23]. A simple URDF model of the gripper (with no joints) was designed in order to simulate the pushing action. Unfortunately, the modelling of the gripper was restricted only to reproducing the gripper when closed, since modelling accurately the gripper (joints and friction) was time consuming. For this reason in the simulation the objects are always supposed to be grasped successfully, for this aim they are removed manually by the user when the robot executes the grasping action. The algorithm was implemented in manner to ask the user the permission to execute the action, this in order to avoid possible dangerous situations, however the possibility to run all automatically has been added. To validate the correctness of the action to execute the RVIZ package[1] was used.

In the simulation the real set up is accurately reproduced (Figure 5.2a), a simulation of the planning system is depicted in Figure 5.2 for a simple problem. The first plan returned is: `(push_dir1 o2) (push_dir1 o0) (grasp o2) (grasp o1) (grasp o0)`. While the real executed plan is: `(push_dir1 o2) (grasp o2) (push_dir1 o0) (grasp o1) (grasp o0)`. The difference is only that it swaps the second action with the first one, this because the two plans have the same length, and at every new frame the system replans again considering it as a separated problem. For the same reason, as long the plan is executed the labels of the objects could change at every new frame.

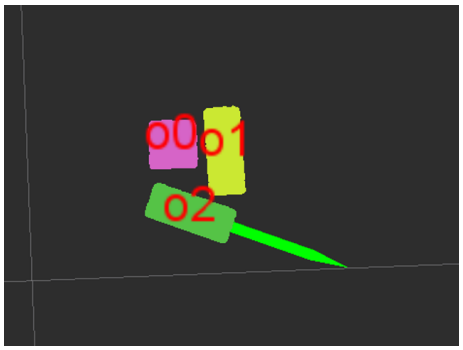
After the algorithm was asserted to work as expected in simulation we moved to perform experiments with real robot.



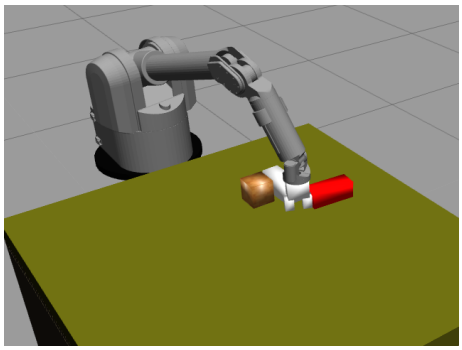
(a) Simulation of the real set up



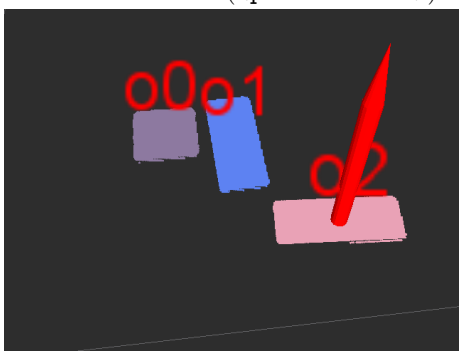
(b) Kinect's view



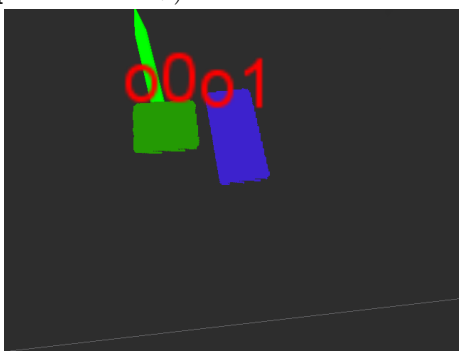
(c) Visualization in RVIZ of the objects and the action to execute (`(push_dir1 o2)`).



(d) Execution of the first plan's action (`(push_dir1 o2)`).



(e) Visualization in RVIZ of the objects and the action to execute (`(grasp o2)`).



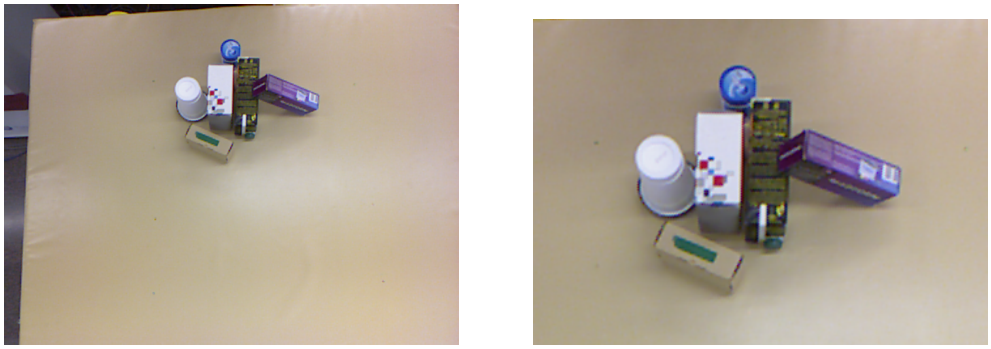
(f) Visualization in RVIZ of the objects and the action to execute (`(push_dir1 o0)`).

Figure 5.2: Simulation in Gazebo of a simple experiment.

6. Experiments

In this chapter some experiment are presented in order to assert the quality of the proposed planning system. The experiment was designed in manner to present the advantages of the planner, i.e. the importance of the pushing action, for several scenarios. Moreover the objects have been chosen accordingly to the gripper's opening width. In particular to include also cases in which the robot fail in grasping an object, we added the set of objects an object which has a width of 6.5 centimetres. Considering the opening width of the gripper is 7 centimetres, it is a difficult object to grasp since the gripper has to be centred perfectly with respect the object.

First Experiment The first experiment presents all the challenges the planners can hold, which are, objects on top of others and objects that need to be moved in order to grasp all of them. We first present the results of a run, next we comment the results of the planning



(a) **(b)**

Figure 6.1

system obtained from 5 runs of this experiment.

7. Conclusions

7.1. Conclusions

7.2. Limitations

The planner here presented has some clear limitations, these are related to the lack of the probability and mainly to the lack of geometric information. How previously said, the pushing action is supposed to push the object far enough from the others ones, this will be true only if the pushing will be performed up to infinite, and this is not the case.. The pushing is performed taking into account the geometry of the manipulated object, without taking care about the surrounding objects geometry and poses. This problem is though to be resolved by replanning, that is, if an object has been pushed but not enough, the planner will return a sequence where the first action is again to push that object. This is actually what the planner does, but since for the most of cases, pushing along directions 1 and 2 is feasible, for the planner both directions are feasible to resolve the problem. So what it could happen is that the planner could first return as solution to push a certain object along direction 1, and then, when replanning, to push the object along direction 2. What is here commented is a infinite loop. This happens because the planner has no information between consecutive frames, that is, it consider the problem as a separated one from the previous frame. This fact is a very undesirable one and there are several scenarios that can present a behaviour like that one. The promising part is the combination of the grasping and pushing actions. It has been observed that the majority of times, although for complex scenarios, the solution is mainly based on a proper sequence of grasping action, while the pushing action is an auxiliary action which is rarely used. Taking this into account, and the presented limitation, is very likely that once an object has been pushed, ones of the objects which cannot be grasped before can now be grasped, avoiding such undesired infinite loop.

Somewhere talk about the limitation of the segmentation that is we assume it to be perfect because if not the planner will return likely an infeasible plan

7.3. Future Work

Bibliography

[1] Rviz http://wiki.ros.org/rviz, available on may 2016.

[2] PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[3] Nichola Abdo, Henrik Kretzschmar, Luciano Spinello, and Cyrill Stachniss. Learning manipulation actions from a few demonstrations. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1268–1275. IEEE, 2013.

[4] Francesco Amigoni, Andrea Bonarini, Giulio Fontana, Matteo Matteucci, Viola Schiaffonati, Aamir Ahmad, Iman Awaad, Jakob Berghofer, Rainer Bischoff, et al. General evaluation criteria, modules and metrics for benchmarking through competitions. 2014.

[5] Julien Bidot, Lars Karlsson, Fabien Lagriffoul, and Alessandro Saffiotti. Geometric backtracking for combined task and motion planning in robotic systems. *Artificial Intelligence*, 2015.

[6] Peter Brook, Matei Ciocarlie, and Kaijen Hsiao. Collaborative grasp planning with multiple object representations. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2851–2858. IEEE, 2011.

[7] Rui Coelho and Alexandre Bernardino. Planning push and grasp actions: Experiments on the icub robot.

[8] Nikolaus Correll, Kostas E. Bekris, Dmitry Berenson, Oliver Brock, Albert Causo, Kris Hauser, Kei Okada, Alberto Rodriguez, Joseph M. Romano, and Peter R. Wurman. Lessons from the amazon picking challenge. *CoRR*, abs/1601.05484, 2016.

[9] N. Covallero and G. Alenyà. Grasping novel objects. Technical Report IRI-TR-16-01, Institut de Robòtica i Informàtica Industrial, CSIC-UPC, 2016.

[10] Richard Dearden and Chris Burbridge. An approach for efficient planning of robotic manipulation tasks. In *ICAPS*. Citeseer, 2013.

[11] Richard Dearden and Chris Burbridge. Manipulation planning using learned symbolic state abstractions. *Robotics and Autonomous Systems*, 62(3):355 – 365, 2014. Advances in Autonomous Robotics — Selected extended papers of the joint 2012 {TAROS} Conference and the {FIRA} RoboWorld Congress, Bristol, {UK}.

[12] Mehmet Dogar and Siddhartha Srinivasa. A framework for push-grasping in clutter. In Nick Roy Hugh Durrant-Whyte and Pieter Abbeel, editors, *Robotics: Science and Systems VII*. MIT Press, July 2011.

[13] Mustafa Ersen, Melodi Deniz Ozturk, Mehmet Biberci, Sanem Sariel, and Hulya Yalcin. Scene interpretation for lifelong robot learning. In *Proceedings of the 9th international workshop on cognitive robotics (CogRob 2014) held in conjunction with ECAI-2014*, 2014.

[14] Mustafa Ersen, Sanem Sariel Talay, and Hulya Yalcin. Extracting spatial relations among objects for failure detection. In *KIK@ KI*, pages 13–20, 2013.

[15] David Fischinger, Astrid Weiss, and Markus Vincze. Learning grasps with topographic features. volume 34, pages 1167–1194, 2015.

[16] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26:191–246, 2006.

[17] Tucker Hermans, James M. Rehg, and Aaron F. Bobick. Guided pushing for object singulation. In *IROS*, pages 4783–4790. IEEE, 2012.

[18] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[19] Dov Katz. *Interactive perception of articulated objects for autonomous manipulation*. University of Massachusetts Amherst, 2011.

[20] Dov Katz, Moslem Kazemi, J. Andrew (Drew) Bagnell, and Anthony (Tony) Stentz . Clearing a pile of unknown objects using interactive perception. In *Proceedings of IEEE International Conference on Robotics and Automation*, March 2013.

[21] Dov Katz, Arun Venkatraman, Moslem Kazemi, J Andrew Bagnell, and Anthony Stentz. Perceiving, learning, and exploiting object affordances for autonomous pile manipulation. *Autonomous Robots*, 37(4):369–382, 2014.

[22] Thomas Keller and Patrick Eyerich. Prost: Probabilistic planning based on uct. In *ICAPS*, 2012.

[23] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE.

[24] Andrey Kolobov, Mausam, and Daniel S Weld. Lrtdp vs. uct for online probabilistic planning. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[25] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, May 2006.

[26] Iain Little, Sylvie Thiebaux, et al. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, 2007.

[27] Bhaskara Marthi, Stuart J Russell, and Jason Wolfe. Angelic semantics for high-level actions. In *ICAPS*, pages 232–239, 2007.

[28] David Martínez, Guillem Alenya, and Carme Torras. Planning robot manipulation to clean planar surfaces. *Engineering Applications of Artificial Intelligence*, 39:23–32, 2015.

[29] Lorenz Mösenlechner and Michael Beetz. Using physics- and sensor-based simulation for high-fidelity temporal projection of realistic robot behavior. In *AIPS*, 2009.

[30] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[31] Melodi Ozturk, Mustafa Ersen, Melis Kapotoglu, Cagatay Koc, Sanem Sariel-Talay, and Hulya Yalcin. Scene interpretation for self-aware cognitive robots. 2014.

[32] Jia Pan, Sachin Chitta, and Dinesh Manocha. Fcl: A general purpose library for collision and proximity queries. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3859–3866. IEEE, 2012.

[33] Jeremie Papon, Alexey Abramov, Markus Schoeler, and Florentin Wörgötter. Voxel cloud connectivity segmentation - supervoxels for point clouds. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, Portland, Oregon, June 22-27 2013.

[34] Morgan Quigley, Ken Conley, Brian P. 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*, 2009.

[35] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[36] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *International Conference on Robotics and Automation*, Shanghai, China, 2011 2011.

[37] Radu Bogdan Rusu, Zoltan Csaba Marton, Nico Blodow, Mihai Dolha, and Michael Beetz. Towards 3d point cloud based object maps for household environments. *Robotics and Autonomous Systems*, 56(11):927–941, 2008.

[38] S. C. Stein, M. Schoeler, J. Papon, and F. Woergoetter. Object partitioning using local convexity. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[39] Andreas ten Pas and Robert Platt. Using geometry to detect grasp poses in 3d point clouds. In *International Symposium on Robotics Research (ISRR)*, September 2015.

[40] Nikolaus Vahrenkamp, Tamim Asfour, and Rudiger Dillmann. Robot placement based on reachability inversion. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1970–1975. IEEE, 2013.