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

June 2016

Abstract

Manipulation planning is a field of study with increasing interest, it combines manipulation skills and an artificial intelligence system that is able to find the optimal sequence of actions in order to solve manipulation problems. It is a complex problem since involves a mixture of symbolic planning and geometric planning. To complete the task the sequence of actions has to satisfy a set of geometrical restrictions.

In this thesis we present a planning system for clearing a table with cluttered objects, which reasons at a symbolic level and reasons about geometric constraints through backtracking. We show that the planning system developed is able to handle efficiently the task.

The main contribution of this thesis is a planning system able to solve a wider variety of scenarios for clearing a table with cluttered objects. Grasping actions alone are not enough, and pushing actions may be needed to move an object to a pose in which it can be grasped. The planning system presented here can reason about sequences of pushing and grasping actions that allow a robot to grasp an object that was not initially graspable.

This work shows that 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 system is a reduction in execution time and it is also easy to implement.

Contents

Chapter 1

Introduction

Robotic manipulation of objects is an increasing field of research which has captured the interest of researches from many years ago. In several industrial environments robots can be easily programmed when the objects are known a priori, i.e. the manipulation is always the same, and robot operations avoid cluttered scenes, but the workspace has to be designed in a manner to provide to the robot a non cluttered scene. However, 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* [**?**], which provides 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 the ones human operators face in Amazon's warehouse. Joey Durham from Amazon Robotics describes the challenges of this competition as follows:

> "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 investigate a simple approach for a complex manipulation problem. Our approach tries to replicate human reasoning during the manipulation of cluttered objects in table clearing tasks. The objective is removing all the objects on a table.

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

This work is structured as follow. First an introduction to the problem we want to solve is presented in Chapter 1. Then, in Chapter 2 a review of the current state of the art in manipulation planning is done. The planning system developed and the algorithm to generate the states will be explained in Chapters 3 and 4 respectively. In Chapter 5 the software design will be presented as well the algorithm's structure. Finally experiments and conclusions about this work are discussed in Chapters 7 and 8 respectively.

This chapter introduces the problem that we are tackling and the experimental set up.

1.1 Problem Approach

In this section the approach to solve the planning problem is described. The strategy to solve the problem is inspired by the way humans solve it. A human would use mainly two actions: grasping and pushing. 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 interacts with the scene to make the object graspable. However, humans are characterized by a high dexterity and therefore they have a lot of freedom in the way to grasp objects. Robots, normally, do not have such a dexterity and grasping in cluttered scene could be very hard without moving the objects.

The pushing action requires to put the end effector in a certain pose and then push the object by moving the end effector. However, it is difficult to push an object and ensure that it follows the desired path since this action depends on its shape. Moreover a controller would be needed in order to correct the pose of the end effector along the path. We assumed that all objects had basic shapes, so that a simple pushing action performs well.

Based on these considerations, the actions the robot has to use are grasping and pushing. Grasping is the most important action since it lets to take an object 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 [**?** **?**]. The pushing is useful when 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 ungraspable. The pushing action can separate adjacent objects that mutually exclude themselves from being grasped. For simplicity we only consider actions that interact with a single object.

The robot's decision maker uses a planning system that returns a sequence of actions that achieve the goal of clearing the table. The robot reasons in an abstraction level by considering only symbolic

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

Figure 1.1: Robot and vision sensor.

predicates with limited geometrical information. Then the plan is checked to see if it is feasible, and if it isn't, backtracking is done.

We assume the world is perfectly known. As actions are actually non-reliable, the planner replans after the execution of each action.

To summarize, in this master thesis we are going to solve the problem of clearing a table with cluttered objects by using a task planner. To do so, a perception system is developed in order to segment the objects through a vision system, and from those objects the states are generated and the planner returns the action to execute. The planning system will generate the states after the execution of each action by receiving a new image from the vision sensor and it plans again.

1.2 Set Up

The set up of the environment the robot will work in is presented here.

The robot used is a Barret WAM arm, which is a 7 degree of freedom (DoF) manipulator arm (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).

To manipulate the objects the robot has 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 its opening width[2] is of 7 centimetres, therefore we are constrained to grasp objects with a width in the range $[3 \div 7]cm$.

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

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

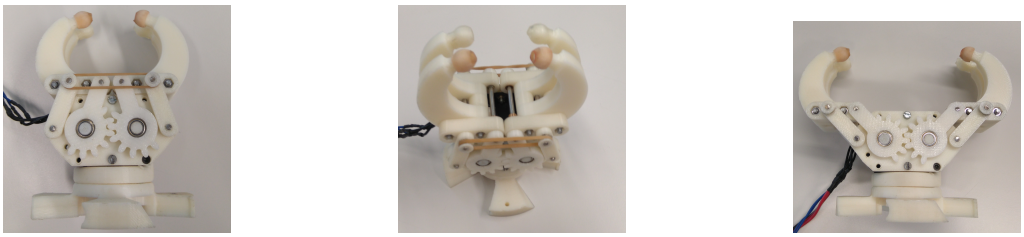


Figure 1.2: Gripper used for the experiments

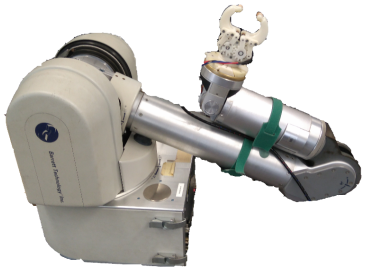
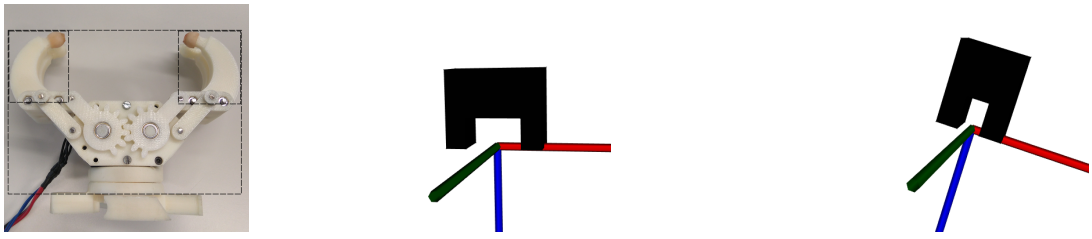


Figure 1.3: Gripper and WAM.

For the task planner, as the reader will see in Chapters 3 and 4, the model of the gripper will be an important resource in order to compute the predicates.

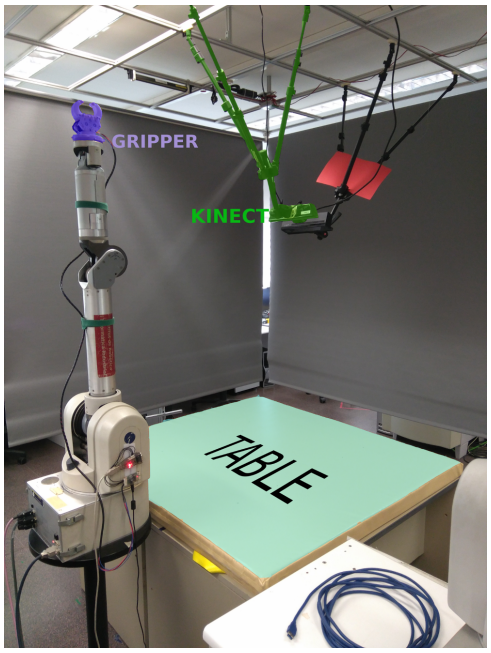
The gripper will be modeled measuring some principal elements such as: finger's width and height, gripper's width, height and deep, closing and opening width. The modeling procedure is depicted in Figure 1.4. The resulting model is a simple triangle mesh which includes all the important geometric information of the gripper. Such a simple model allows the collision algorithm commented in Chapter 4 to check for collision in just a few milliseconds. A more detailed and complex model would have higher precision, but such a high accuracy is not needed, and it would slow down the algorithm. The gripper is mounted in the end effector of the robot as shown in Figure 1.3.

The scenario the robot is going to work in is composed of a table and the objects will lay on top of



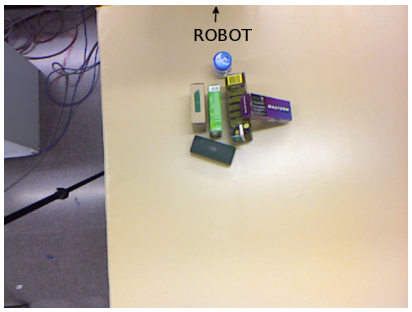
Elements measured Opened gripper mesh model Closed gripper mesh model

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.



(a) Principal elements of the experimental set up.

(b) Example of a cluttered scene.



(c) Kinect's view.

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

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. Figure 1.5b shows an example of a cluttered scene the robot is going to deal with, and Figure 1.5c shows the same scene as seen by the Kinect. To avoid occlusions, we wait until the robot finishes to execute an action and moves away before taking new images.

Chapter 2

Previous works

In this chapter we present some previous works regarding manipulation planning. In the other chapters the state of the art will be introduced more in concrete accordingly to the chapter's topic.

Many manipulation planning approaches[**?**] 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. They require to obtain the symbolic predicates that represent geometric features, which are very time consuming. Therefore, these hybrid planners can be too slow in real applications.

Dogar and Srinivasa [**?**] proposed a framework for planning with cluttered scenes using a library of actions inspired by human strategies. They designed a planning system that decides which objects to move, the order, where to move them, and the appropriate manipulation actions. Moreover, it accounts for the uncertainty in the environment all through this process. The planning system first attempts to grasp the goal object, and if it is not possible, it identifies what is the object that prevents the action and adds it to a list of objects that have to be moved. Afterwards, those objects are moved in whatever position 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. Our approach performs better with goals that involve more than one object. We plan sequence of actions considering all objects in the goal. The actions they use to move objects that were in the way may actually hinder future goals.

A recent alternative proposed by Mösenlechner and Beetz [**?**] 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, the effects of actions are determined by simulating them, and then the mapping is used to update the symbolic state. However, their method requires to know the physic of the manipulated objects to simulate them. Moreover the authors didn't test their planning system with a complex scene like the ones faced in this thesis. Our planning system doesn't use any simulator, instead it relies on a prediction algorithm to represent how the objects can be manipulated, leading to a faster and easier to implement solution.

In [**?**] the authors address a problem similar to the one of this thesis. The authors blended pushing and grasping actions for a table manipulation task. They use the concept of reachability [**?**] to exclude impossible poses of the gripper at the planning stage, creating a reliable plan suitable for real-time operation. The authors model the planning problem through a Markov Decision Process (MDP), discretizing the world in grid cells and assigning each one a push and grasp vector defined by the reachability concept. Their advantage is that they plan a trajectory level so they can consider more details. In contrast, we plan at an action level, so we can consider more complex goals involving several objects, and will optimize the sequence of actions for completing the whole task. Moreover, while their method needs to be adapted to each robot, to build a reachability map, our method can be directly integrated in any robotic manipulator.

Symbolic planning requires knowledge about the preconditions and effects of the individual actions and such a knowledge can be obtained through machine learning techniques. In [**?**] the authors proposed an approach to learn manipulation skills, including preconditions and effects, based on teacher demonstrations. With just a few demonstrations the method learns the preconditions and effects of actions. This work looks promising since it allows to resolve planning problem by learning the model, but it is suitable only for simple actions. Having a hand-built model, like the one of our work, lets to resolve more complex problems and also it is more straightforward.

In [**?**] Dearden and Burbridge proposed an approach for planning robotic manipulation tasks which uses a learned bidirectional mapping between geometric states and logical predicates. First, the mapping is applied to get the symbolic states and the planner plans symbolically, then the mapping is applied to generate geometric positions which are used to generate a path. If this process fails they allow the system a limited amount of purely geometric backtracking before giving up and backtracking at the symbolic level to generate a different plan. However, this method cannot tackle complex scenes, such as cluttered objects, since in those cases learning a good mapping would be very hard.

Compared to the state of the art, we propose a planning system for clearing cluttered objects.

Our approach plans at a symbolic level, which is efficient and is low time consuming (the time to get a plan is usually less than 0.5 seconds). As far as we know, previous approaches haven't tackled very cluttered scenes, such as the one in Figure 1.5b. We will also show that the lack of geometric constraints introduces some limitations to the system, but the general results obtained are good and the planning phase is very efficient.

Chapter 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...) [**?**]. A very well known classic planner is the Fast Downward planner [**?**].

Hierarchical planning, also called *Hierarchical Task Network*(HTN), works in a similar way to how it is believed that human planning works [**?**]. 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 [**?**].

Probabilistic planning is a planning technique which considers that the environment is not deterministic but probabilistic. So the actions have a probability to obtain a certain state, and given an initial state and a goal state, the planner finds the solution path with the highest reward, which depends also on the probability. Probabilistic problems are usually formalized as Markov Decision Processes (MDP). In this category two probabilistic planners that performed good in planning competitions are Gourmand [**?**] and PROST [**?**].

3.2 Planner

The problem involves a big amount of uncertainty due to the interaction of the robot with the environment. When the robot interacts with the objects, it is very hard to predict correctly the position of the object after the execution, that is the next state. A probabilistic planner considers the probability associated with every effect to predict the state after executing an action, such a probability has to be specified or learned for each type of object.

Martinez et al. [**?**] faced the problem of cleaning a surface entirely by probabilistic symbolic planning. The problem they solved was characterized by a strong uncertainty, and a unexpected effect of an action would require to replan and therefore to slow down the system. Another way to face the problem of probability is replanning after each executed action [**?**], or whenever the current state deviates from the expected one, generating a new plan from the current state to the goal. In this case the actions are considered deterministic, and only the first action is executed before replanning again.

Little et al. discussed in [**?**] 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, in many problems it is more efficient to replan with a deterministic planner rather than directly using a probabilitic planner.

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 by a deterministic planner.

A hierarchical planner would be a good choice if the problem presented some hierarchies, for instance in the case the goal was to clear several tables. Since the problem is about cleaning a single table it is more straightforward to use a classical planner.

The planner chosen was the **Fast Downward** planner [**?**], a very well know classic one. This planner is feature-wise complete, stable and fast in solving planning problems.

3.3 Symbolic Predicates

3.3.1 Formulation

The problem is formulated as a 6-tuple $\Pi = \langle S, s_o, G, A, T, c \rangle$, where:

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

The plans 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 $C(\tau_i) = \sum_{a \in \tau} c(a)$. The optimal solution is the solution with less cost: $\tau^* = \min_{\tau_i} c(\tau_i)$. In this work we do not specify specific costs for the actions, so the planner returns the plan with fewer actions.

3.3.2 Predicates

The *Fast Downward* planner needs the problem to be formulated in *Problem Domain Description Language* (PDDL) [**?**]. 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 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 symbolic predicates are added so that the planner can reason about collisions and geometrical constraints.

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:

- **removed**: `(removed o1)` means that object `o1` has been grasped and removed from the table. The goal is reached when all the objects have been removed.
- **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 would grasp it, the object above would 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 object `o2` 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 so 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 it is because the object would collide, or the gripper would collide.

- **ik_unfeasible_dir1**, **ik_unfeasible_dir2**, **ik_unfeasible_dir3**, **ik_unfeasible_dir4**, **ik_unfeasible_grasp**: to consider the geometrical constraints regarding the working space of the robot, a predicate which states whether the inverse kinematic has solution is added for each action. For instance, `(ik_unfeasible_dir1 o1)` means that the inverse kinematic to push the object `o1` along direction 1 has no solution.

3.3.3 Actions

In this section the actions preconditions and effects are described.

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 grasp the desired object,
- the inverse kinematic has solution.

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

- the grasped object is removed and 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.

3.3.4 Backtracking

The geometrical constraints related to the inverse kinematic of the robot are computationally 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[**?**]. Backtracking is based on a two-fold strategy:

1. checking if the plan is feasible,
2. if it is not, the state is updated with the new information and the system repeats from point 1 until a feasible plan is obtained.

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 and goal state G.

Outputs: a feasible plan or not plan at all.

```
procedure GETPLAN(s0,G)
    repeat
        plan ← GETFASTDOWNWARDPLAN(s0,G)
        if ¬ISPLANEMPTY(plan) then return NULL
        end if
        action ← GETFIRSTACTION(plan)
        success ← HASIKSOLUTION(action)
        if ¬success then
            s0 ← 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. 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.

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 for pushing them. It has to interact with the white cup in order to make space to move the other objects and then grasp them. For this case the system would perform the following set of operations:

1. It first gets the following plan: `(grasp o2)`, `(push_dir1 o0)`, `(grasp o1)`, `(grasp o0)`,
2. It solves the inverse kinematic for the `(grasp o2)` action, but it finds no solution and adds to

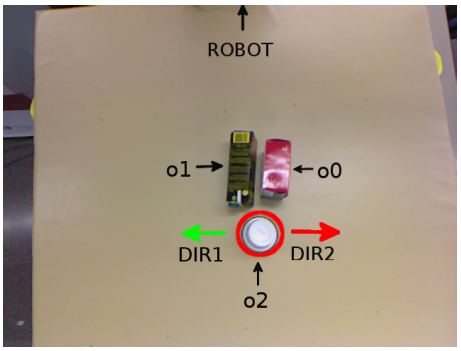


Figure 3.1: Unfeasible plan due to geometrical constraints. In this case the planner returns as action to execute grasping or pushing away the white cup (highlighted by a red circle) but it is out the configuration space of the robot and there exist no plan for that problem.

the states the predicate `(ik_unfeasible_grasp o2)`,

3. It replans and gets the following plan: `(push_dir1 o2)`, `(grasp o2)`, `(push_dir1 o0)`, `(grasp o1)`, `(grasp o0)`,

4. It solves the inverse kinematic for the `(push_dir1 o2)` action but it finds no solution, so the predicate `(ik_unfeasible_dir1 o2)` is added to the states,

5. It continues until there exists no solution for the planning 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 objects 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 the object is removed from the goals so that the rest of the goal can be completed.

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 7.7. It can be appreciated the two fold strategy of the algorithm, the first stage (Figure

3.2a) is devoted to get the predicates from the Kinect sensor and the second one to get a plan, to evaluate its feasibility and to execute it (Figure 3.2b).

3.4 PDDL

For clarity purposes, the PDDL syntax of the described actions is shown here. 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 removed
            (removed ?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)))
              (when (block_dir3 ?o ?x)(not (block_dir3 ?o ?x)))
              (when (block_dir4 ?o ?x)(not (block_dir4 ?o ?x))))
```

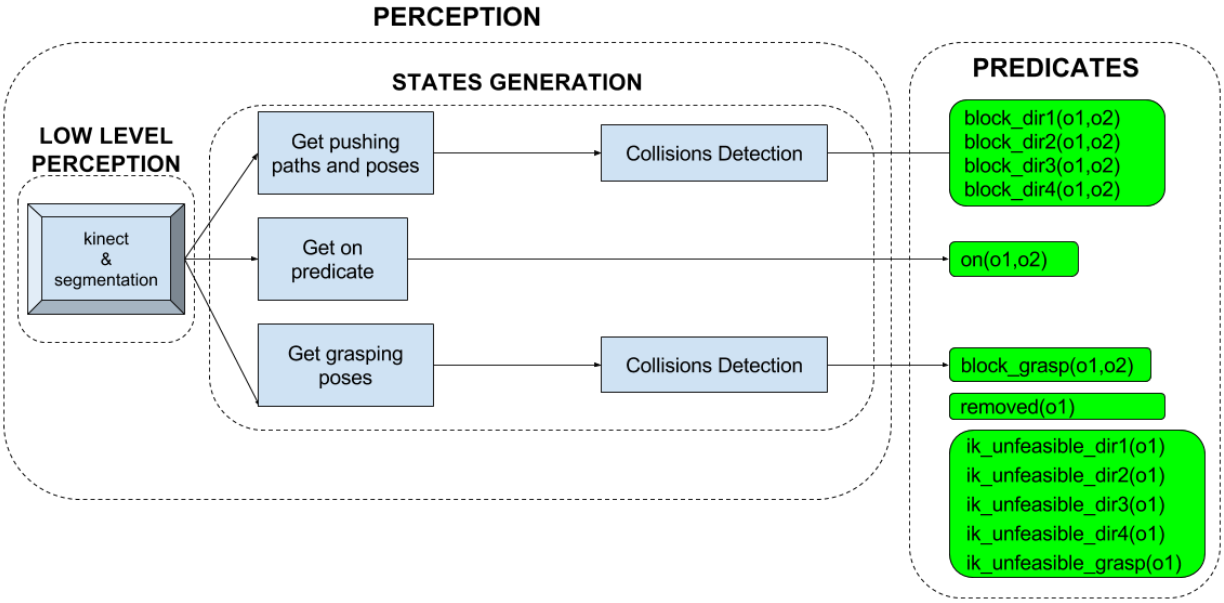
Listing 3.1: PDDL syntax of the grasping action

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

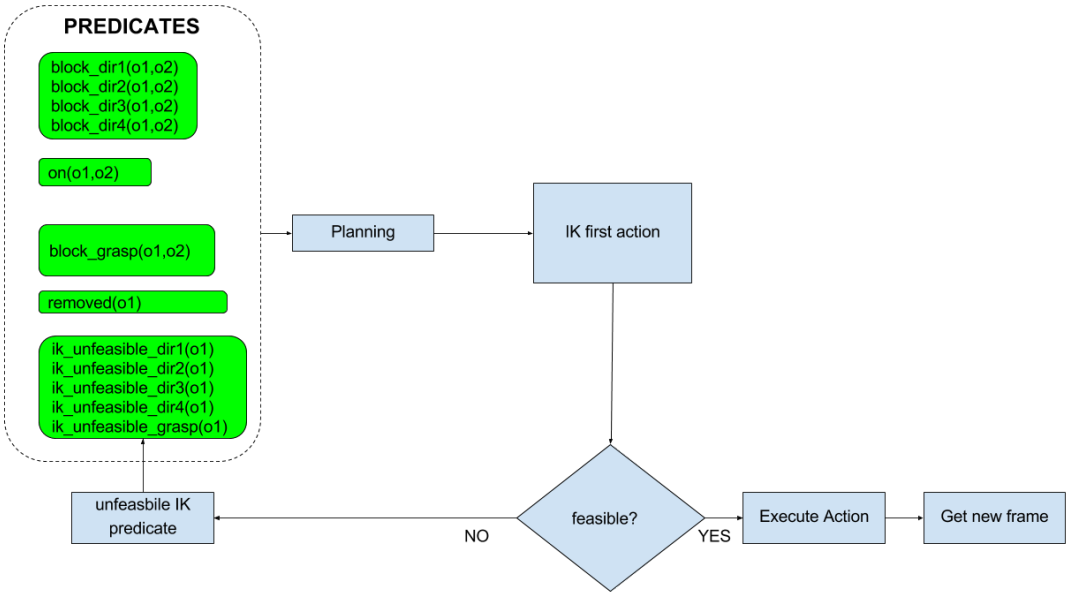
```
(:action push_dir1
```

```
    :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



(a) Perception Pipeline



(b) Planning Pipeline

Figure 3.2: Perception and planning pipeline

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 detection

To interact with the objects we have first to detect them. To do so, 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.

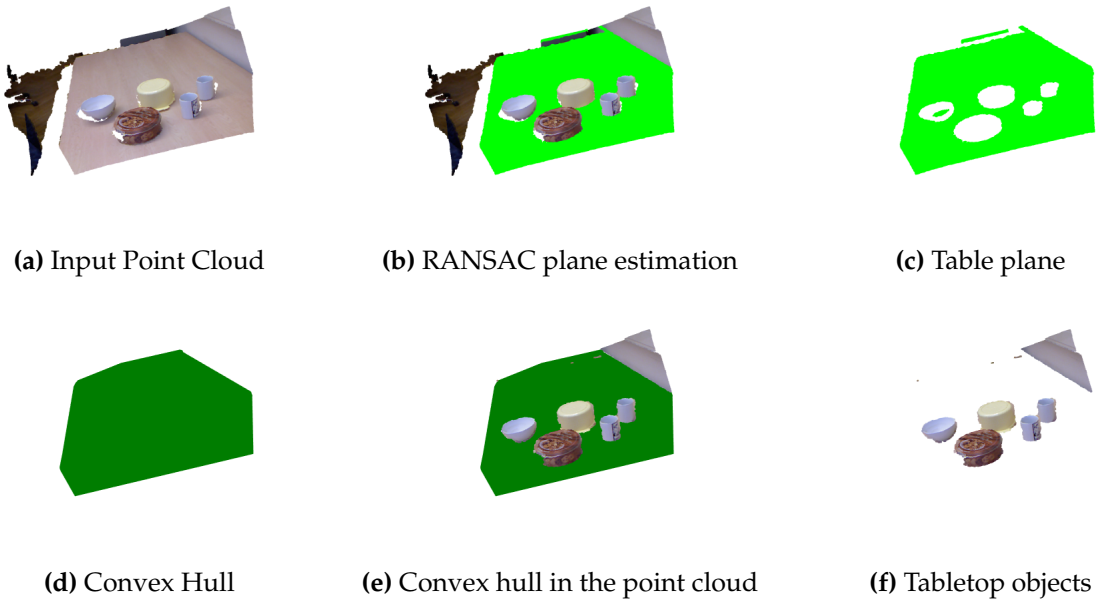
This stage is composed of 3 steps:

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 [**?**]. 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:



(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).

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.

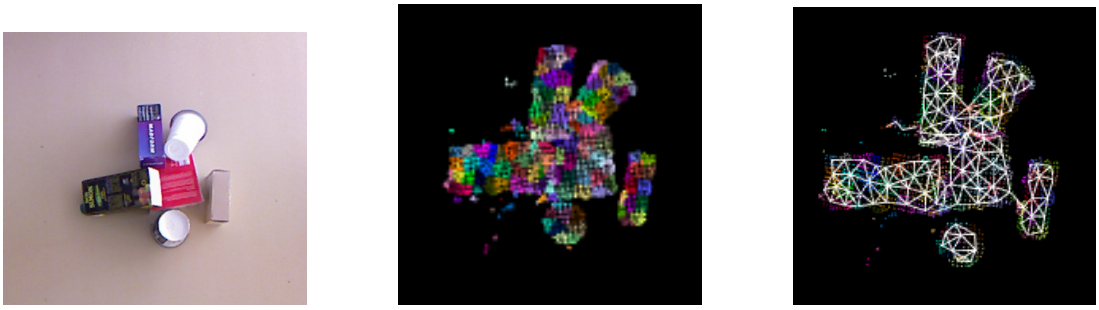
3. **Polygonal prism projection**: all the points are projected on the table plane previously estimated and all the points which projections belong to the 2D convex hull are considered to be points of tabletop objects. The points that do not belong to it are points of non-tabletop objects.

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

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.

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



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

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

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 [**?**], 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.

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). 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.

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

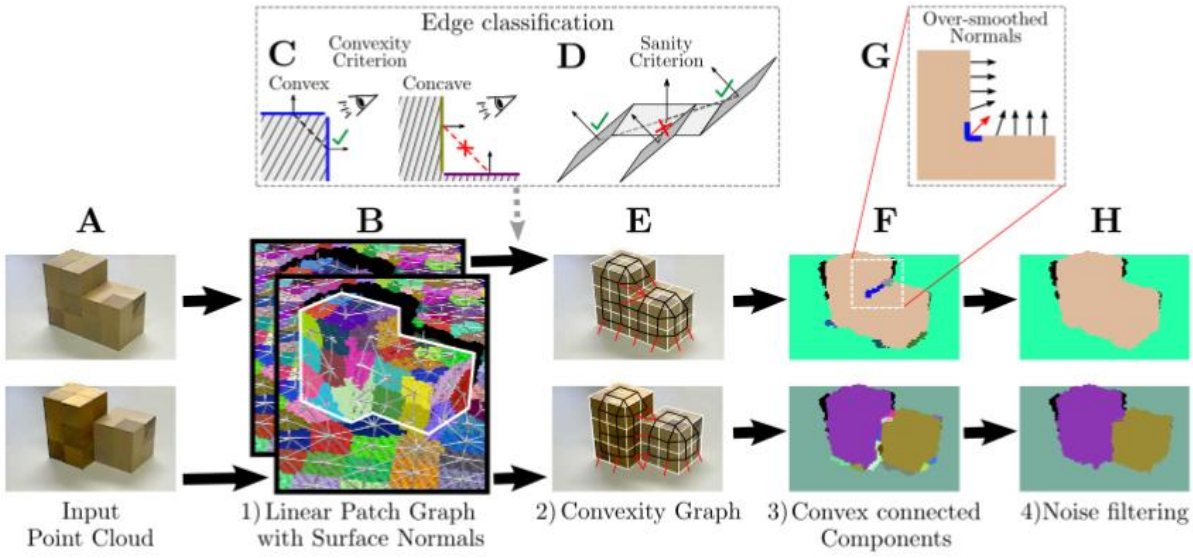


Figure 4.3: LCCP algorithm's structure. Reproduced from [**?**]

Figure 4.4: Example of segmentation results.

- 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.

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

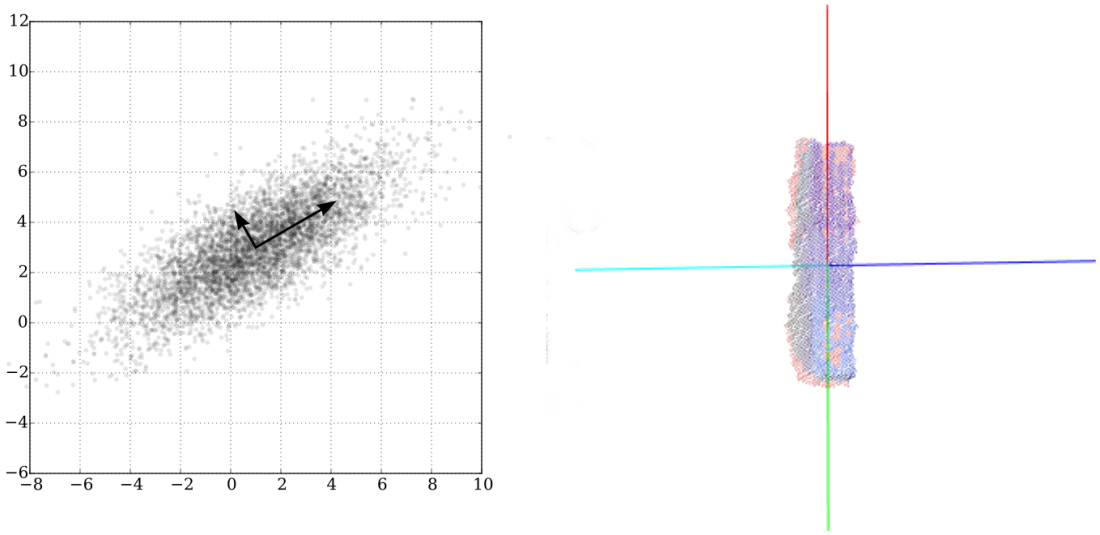
In this section some concepts, that will be used to execute the actions and to generate the states, are presented.

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) [**?**] 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

[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.

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

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

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

1. Calculate the origin point $\mathcal{P}_O = (x_O, y_O, z_O)$ of the plane, which can be calculated by arbitrary x_O and y_O coordinates as
$$z_O = \frac{-1}{c}(ax_O + by_O + d),$$
then 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 upwards.

Rotation Matrices Rotation matrices 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

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

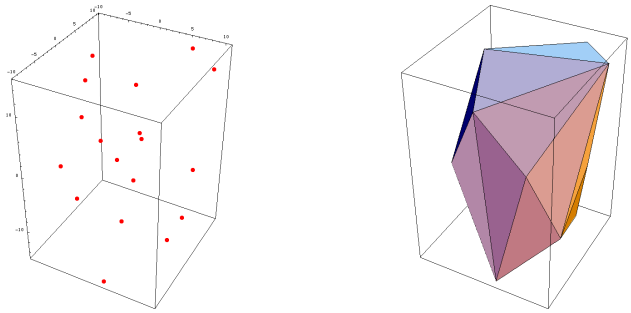
To transform any object, such as the gripper mesh model, to a pose defined by frame $\{B\}$ then the following homogeneous transform is applied:

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

where ${}^{B}_{A}R = {}^{A}_{B}R^{\top}$ 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.

[3] https://en.wikipedia.org/wiki/Bounding_volume



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

Figure 4.7: Convex hull example

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 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.

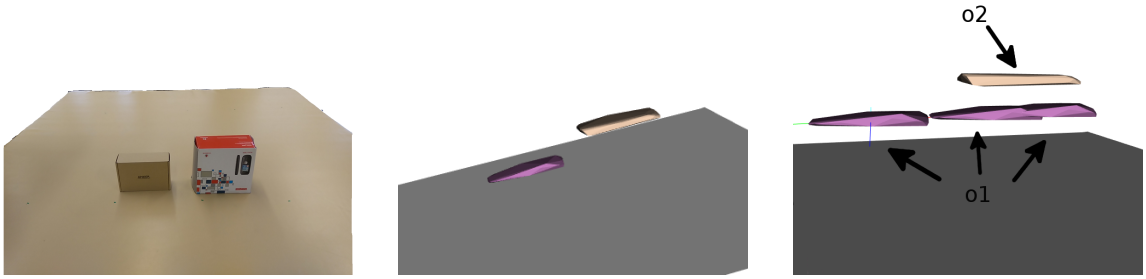
Given two objects A and B and their configurations $\mathbf{q_A}$ and $\mathbf{q_B}$, the collision test returns a boolean value about whether two objects collide or not [**?**]. Two objects collide if

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

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.

[4]Images obtained from `http://xlr8r.info/mPower/gallery.html`



(a) (b) (c)

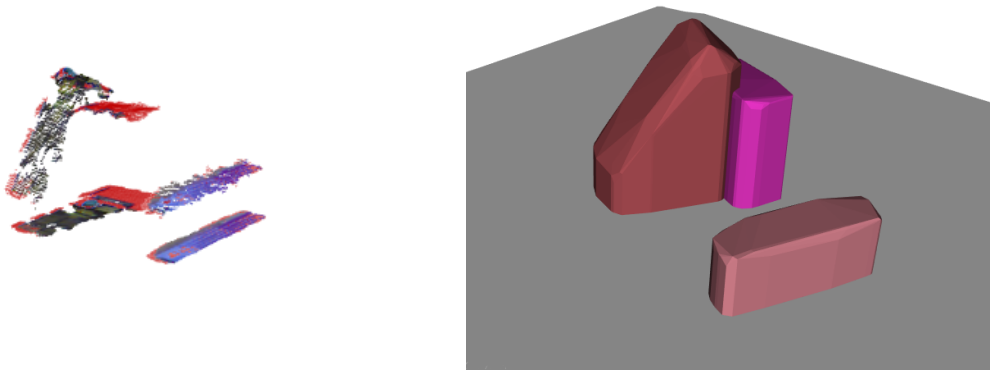
Figure 4.8: Convex hulls and collision detection using the segmented objects retrieved by the LCCP segmentation algorithm. The gray surface represents the plane's 2D convex hull. In Figure 4.8b it is possible appreciating that we miss the information about the hidden part of the object. In Figure 4.8c 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.

Objects Modeling The Kinect can mainly see one face (usually the top) of the objects and therefore we cannot apply directly the convex hull algorithm to the detected surfaces. If we applied the convex hull on an object's observed surface, we would 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 missing the surfaces that cannot be seen from the Kinect's point of view.

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, he would imagine that the objects are not floating surfaces, and he/she would 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 Action 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.



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

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

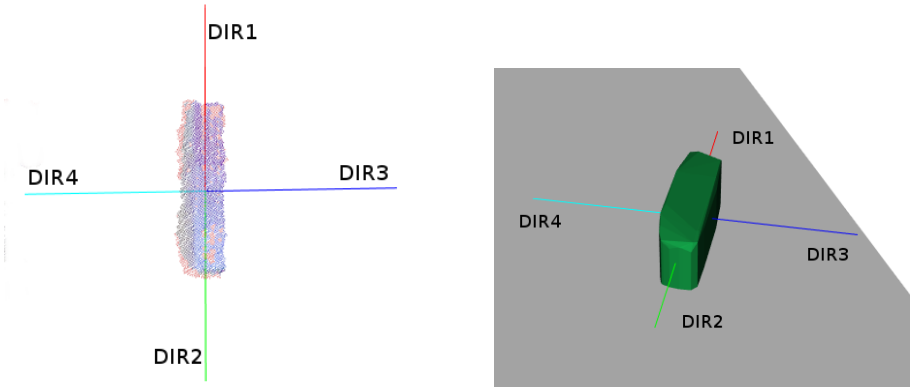
4.3.1 Pushing

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 [**?**] [**?**] [**?**] , without taking care about the final position of the objects or about eventual collisions. Hermans er al. [**?**] 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.

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 its first two principal directions as possible directions to push an object. 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. For this aim the first two principal directions are projected on to the table plane. So the pushing directions considered are not the principal directions but their projections.

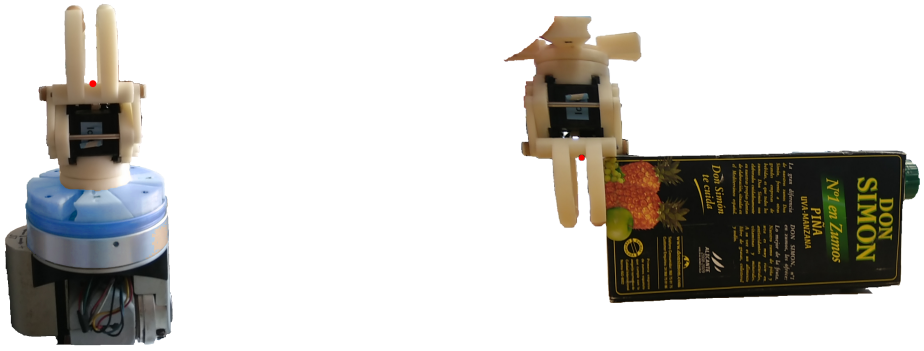
Next, the pose of the gripper is computed accordingly to its shape, to the shape of the objects and the pushing direction.



(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.



(a) The gripper and its base.

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

Figure 4.11

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 depth is less than 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 as accurate as possible, we don't want that the gripper's base touches the manipulated object.

Knowing 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

(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 to 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.

to 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 (7 centimetres) the objects the robot can manipulate are thin. This means that when pushing along the principal axis, the object's width is likely small(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 used is the one in Figure 4.12b. The pose of Figure 4.12a is more stable since the contacts point (the fingers) are more distant between themselves. For this reason this 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:

$$R_{dir1} = \begin{bmatrix} dir2_X & dir2_Y & dir2_Z \\ dir4_X & dir4_Y & dir4_Z \\ n_x & n_y & n_z \end{bmatrix}^\top \qquad T = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \tag{4.1}$$

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 closing point[5].

As previously said, the planner considers to push the objects at infinity, but in reality the robot has to push the objects for a finite length. Since the planner has no geometric information, it does not know how much it has to push in order to singulate the object, or to make it graspable. For this reason the pushing length is chosen accordingly the dimension of the OBB 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 $l = k \cdot OBB_{\texttt{o1}}(length\_object)$, 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 of the surrounding ones. The improvement of this limitation will be one of the topics for future works.

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 calculated. 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!*[6] 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, 20 centimetres from the pushing pose. This pre-pushing 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 we considered another one translated, accordingly the table normal, 20 centimetres from the last pose. In this way the pushing trajectory is defined by a total of $n + 4$ poses.

4.3.2 Grasping

There exist an advanced state of the art regarding grasping. Despite this, all the techniques of grasping are usually 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[**?**]. 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 [**?**] and HAF [**?**],

[5]The gripper's closing point is finger's contact point. (Although in this case the fingers do not touch themselves)

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

despite this, they are not so robust and they are computationally expensive and not suitable for this thesis [**?**]. 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[7] is orthogonal the principal axis of the object. The approaching direction[8] 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. Once the object has been grasped it is easy that it collides with other ones, therefore a post grasping pone is defined by translating the grasping pose for 20 centimetres along the table's normal.

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 the post grasping pose again.
6. Going to the dropping pose: the object will be dropped into a bin.

4.4 States generation

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

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

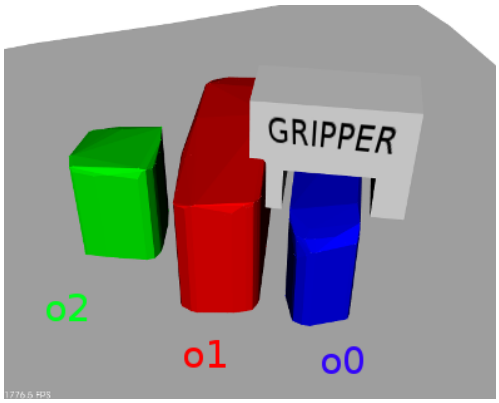


Figure 4.13: Visualization of the computation of `block_grasp` predicate for object `o0`. The opened gripped model is transformed to the grasping pose for object `o0` and it is tested if the gripper mesh model collides with the other objects, in this case it collides with `o1`.

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.

Notice that this method requires 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 [**?**] [**?**] [**?**] the authors proposed some approaches to enhance 3D recognition and segmentation

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 $A \in O$ **do**
    $gripperMeshTransformed$ ← TRASNFORMGRIPPERMODEL($G_{poses}(A)$)
    for all $B \in O$ **do**
      if $A \neq B$ **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

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. 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).

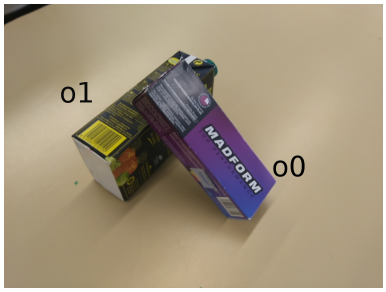
Although this method works fine to compute the `on` predicate it has the limitation that its scope is only for objects for a rectangular shape, or similar shapes.

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:

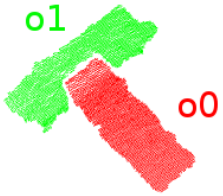
$$\texttt{(on o0 o1)} = \begin{cases} True, & length(C_{P_0} \cap P_1) < th_0 \quad \wedge \quad length(C_{P_1} \cap P_0) > th_1 \\ False, & otherwise \end{cases} \tag{4.2}$$

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 $th_0 = th_1 = 100$.

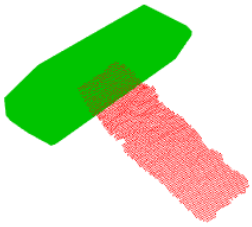
The formula 4.2 is then evaluated for every possible combinations of objects, therefore the complexity to generate this state is $\mathcal{O}(n^2)$, where n is the number of objects. Despite this, its computation is very fast. The example in Figure 4.14 was evaluated just 2 times since there are only 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



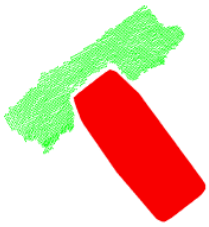
(a) Scene



(b) P_0 & P_1



(c) C_{P_1} & P_0



(d) C_{P_0} & P_1

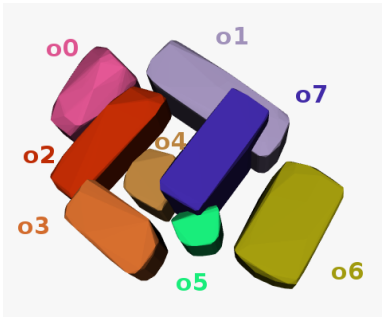
Figure 4.14: Visualization of the computation of the `on` predicate. Figure 4.14b shows the real image, Figure 4.14b shows the projections of the objects onto the table plan while Figures 4.14c and 4.14d represent the two steps strategy to compute the `on` predicate.

time devoted to compute this predicate would be approximatively $10 \cdot 9 \cdot 1.5 \approx 135ms$.

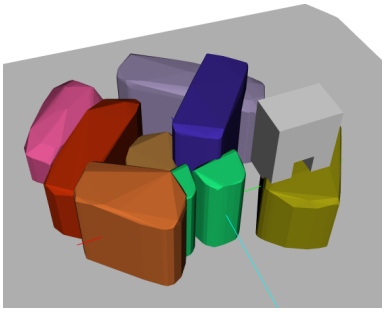
4.4.3 Predicate: `block_dir`$_i$

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

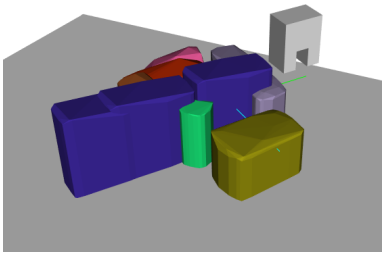
Object `o1` can blocks object `o0` to be moved along a certain direction if a collision will appear between the two objects. In order to do that, having a certain pushing length l_i for the i-th direction of object `o0`, its convex hull C_{o0} is translated along the considered direction until reaching 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.



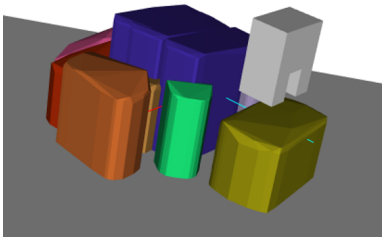
(a) Top view of the convex hull of the segmented objects.



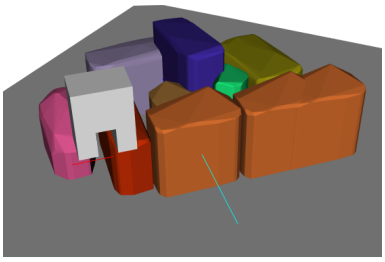
(b) Evaluating the `(block_dir1 * o5)` predicate



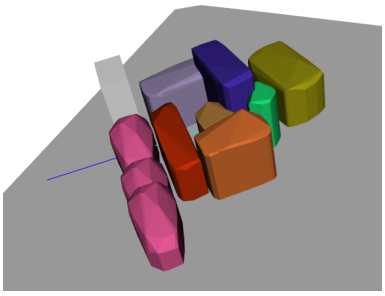
(c) Evaluating the `(block_dir1 * o7)` predicate



(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)` predicate" means that the algorithm is evaluating for all the objects, except `o0`, if they collide with `o0` when pushed along direction 1.

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 fine 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. From that pseudo code can be noted that the complexity of this function to compute the predicate approximatively is $\mathcal{O}(n^2)$, where n is the number of objects.

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 $\approx 0.869 s$ were dedicated to check if the objects would collide when moved, while $\approx 0.239 s$ 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 ← 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

Chapter 5

Software design

In this chapter the design of the software and the external libraries used are briefly described.

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

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 [**?**]. 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, writes the problem in PDDL syntax, calls the Fast Downward binary file and returns a plan,
- a node to evaluate the execute the first action of the plan,

- a decision maker node which controls all the processes and decides the next task to do.

These nodes are implemented as services, 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 the segmented tabletop objects and the plane coefficients. Then it calls the `states.srv` service giving as input the tabletop objects point clouds 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 and a plan is obtained. 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, the decision maker adds to the states the `ik_unfeasible` state for that action and recalls 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. If no plans exist could be fault of the segmentation, therefore if there are objects on the table the algorithm is iterated until the robot can do something.

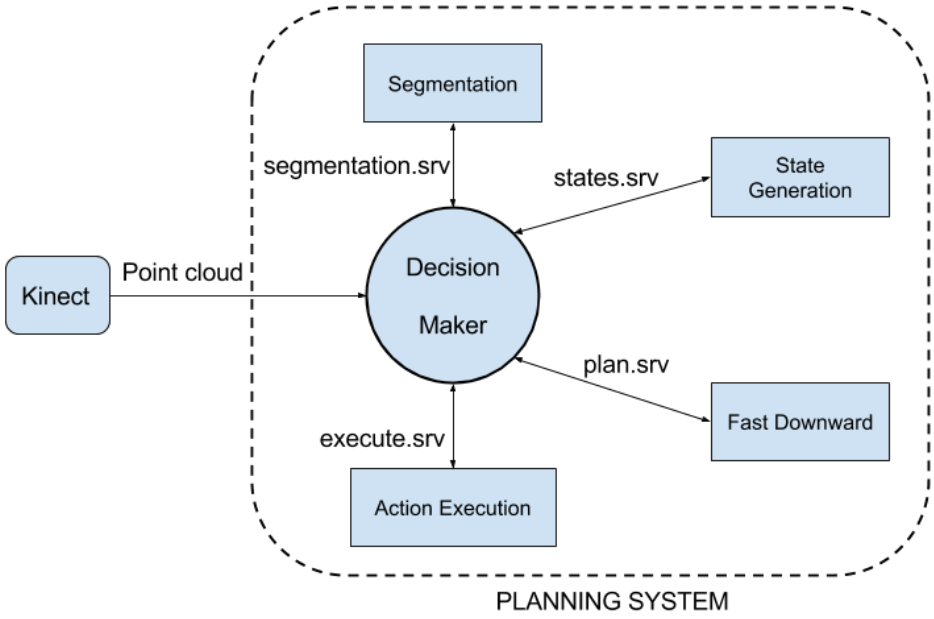
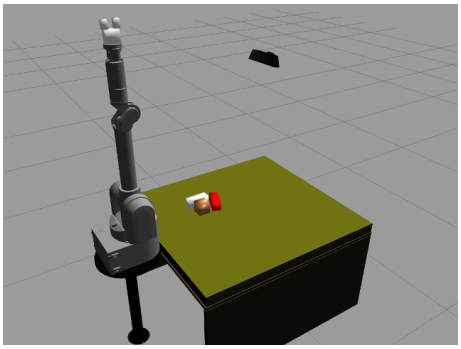


Figure 5.1: Software architecture.

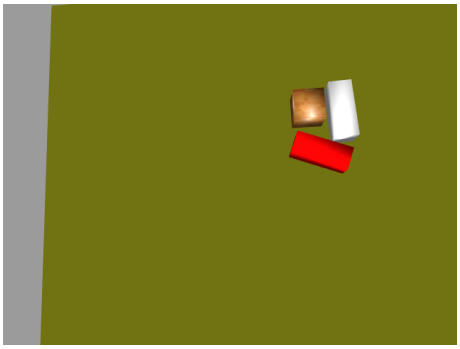
Simulation Before to test the implemented algorithm with the real robot it was first tested in simulation with Gazebo[**?**]. A simple URDF model of the gripper (with no joints) was designed in order to simulate the pushing action. Unfortunately, the modelling of the gripper was restricted only to reproduce the closed gripper model (the modelled gripper cannot open and grasp). 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. To validate the correctness of the action to execute the RVIZ package[**?**] has been 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 third one, this because the two plans have the same length and at every new frame the system replans again considering it as a problem uncorrelated to the previous one. For the same reason, as long the plan is executed the labels of the objects could change at every new frame.

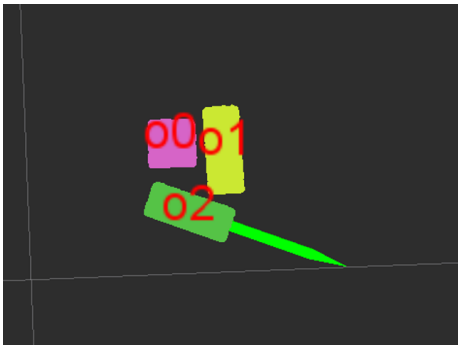
After the algorithm was asserted to work as expected in simulation we moved to perform experiments with real robot.



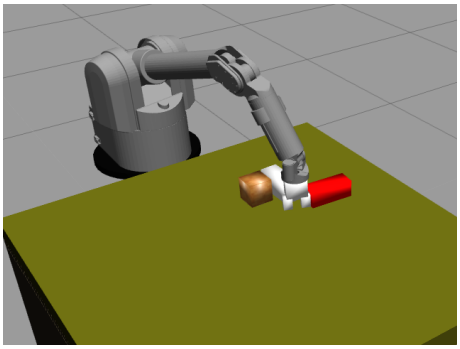
(a) Simulation of the real set up



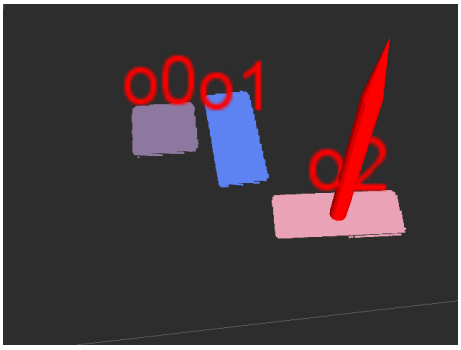
(b) Kinect's view



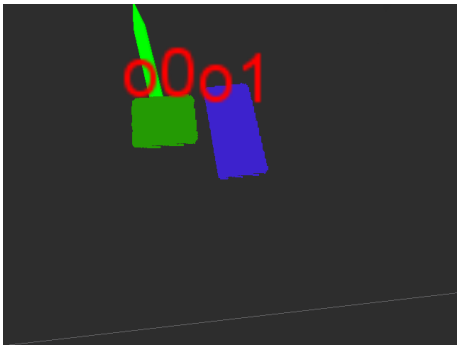
(c) Visualization in RVIZ of the objects and the action to execute (`(push_dir1 o2)`).



(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.

Chapter 6

Project management

This chapter covers economic and environmental analysis

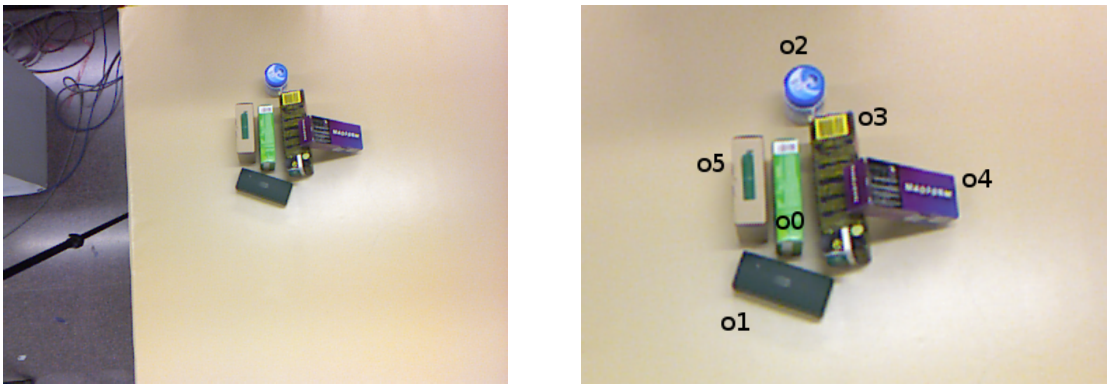
Chapter 7

Experiments

In this chapter an experiment is presented in order to assert the quality of the proposed planning system. The experiment presents the advantage of the blending of pushing and grasping actions.

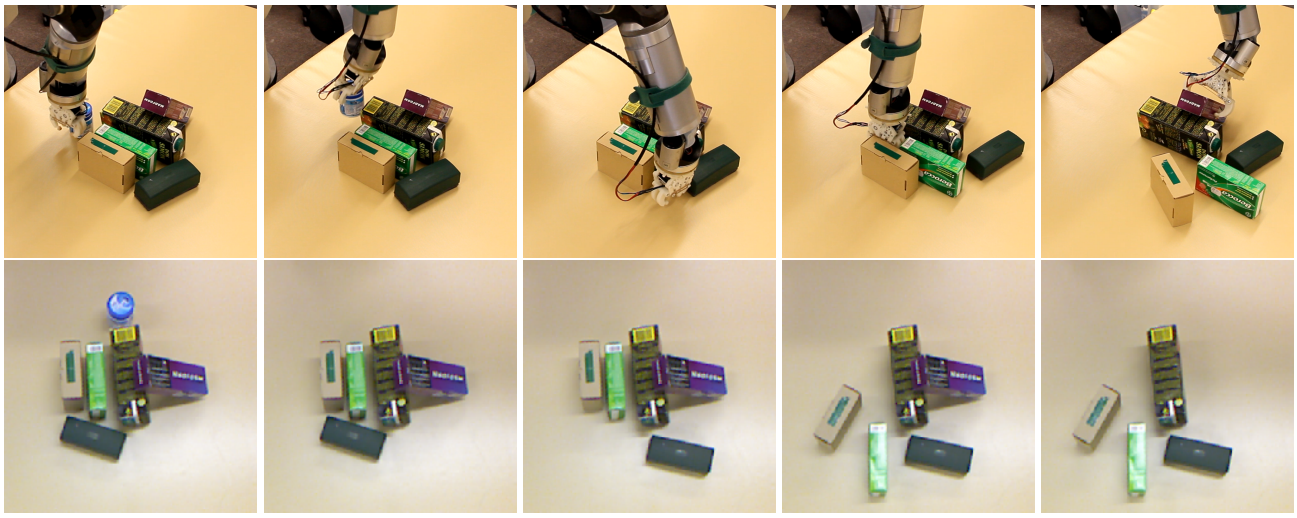
The experiment (Figure 7.1) presents all the challenges the planners can handle, which are: objects on top of others and objects that need to be moved in order to grasp them.

In Figure 7.2 the execution of the plan and the results of each action are shown. The plan first decides to push away object o2 since it is not able to grasp it due to its grasping pose. In that pose the Kinect is able to see also a side of the object, and therefore its second and third principal components make the grasping pose being the one in Figure 7.3, which is colliding which object o3. Therefore it has to move object o3, but it cannot because o2 hinders that action, another option is pushing away o2 and then grasp it. In the next iteration the system takes a new image and the grasping pose of o2 has no collisions, and the system decides to grasp it. Then it pushes away object o1 because it is not



(a) Kinect's view **(b)** Objects labels.

Figure 7.1: Kinect's view of the first experiment.



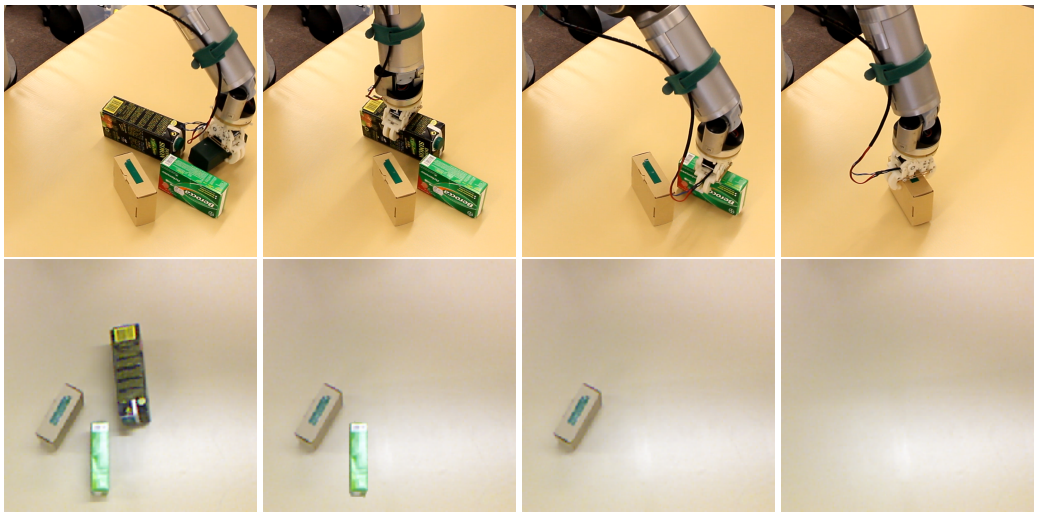
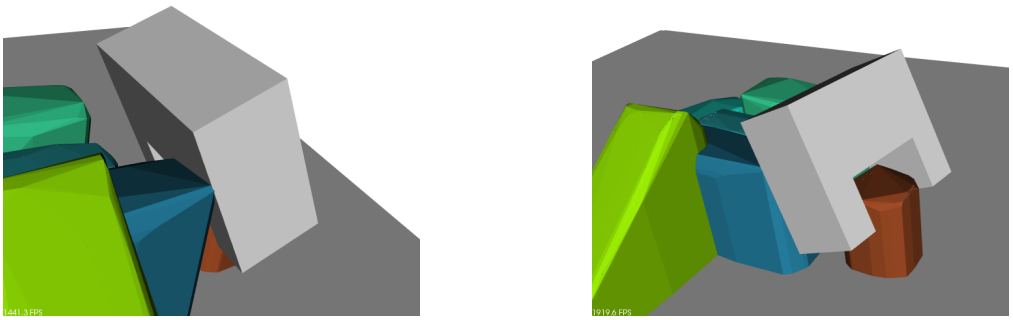


Figure 7.2: Results of a run for the experiment of Figure 7.1. The executed plan is: `(push_dir3 o2) (grasp o2) (push_dir1 o1) (push_dir1 o0) (grasp o4) (grasp o1) (grasp o3) (grasp o0) (grasp o5)`. The first and third row of images show the robot executing the actions, while below the results of those actions.



(a) **(b)**

Figure 7.3: Collision between the gripper and o3 for the grasping pose of object o2.

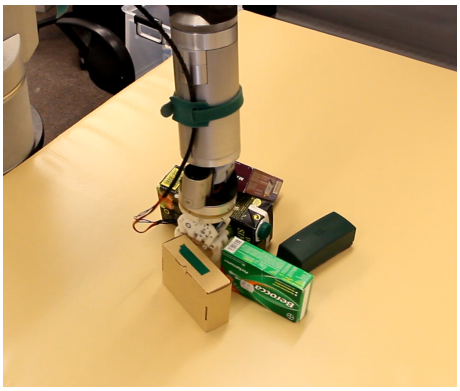


Figure 7.4: Unexpected outcome: pushing o0 the gripper touches also o5.

possible to grasp it or to move the other objects having o1 in such a position. Next it pushes object o0, this action shows the optimality of the planner. If it did not push o0 it had to push away first o5 and o3 to make o0 graspable, it saved one action pushing away o0. The outcome of this action is not the expected one since the gripper, during the pushing action, touched slightly also o5 (Figure 7.4). In this case is possible noting the ability of the system to adapt to unexpected outcomes. Then the robot finishes the task grasping the remaining objects.

In order to get some statistics regarding the planning system we performed 6 runs of the experiment. All the times commented refer about an execution on a machine with dual-core CPU with 3.16GHz. In Table 7.1 the executed plans of all the runs are reported. The plans are different because of two main reasons: the segmentation is not always the same and in the attempt to reproduce the experiment the objects could have been not positioned in the same poses of previous experiments.

From these experiments it has been noted that the segmentation have a strongly importance since, due to the noise of the Kinect, the segmentation was not always the same, despite the filtering. Also the tuning of the segmentation parameters made it quite noisy sensitive. Moreover, at the edges the

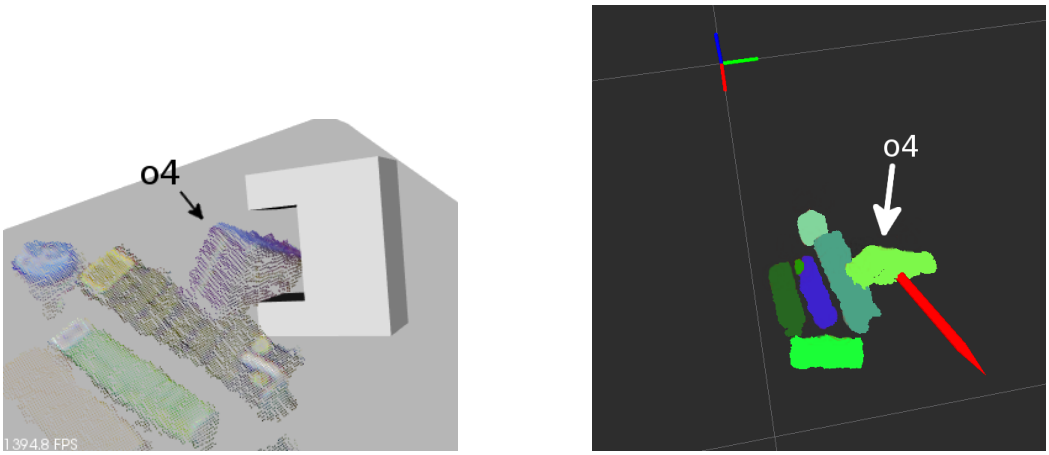


Figure 7.5: Grasping pose of object o4 with unfeasible IK.

segmentation algorithm had some drawbacks since it could cluster some edges points to the biggest adjacent clusters of supervoxels, this means that some edges of an object could be detected as part of the adjacent object. This, in several situation, can leads to find no plan since the object model is not similar to the real object, and during the computation of the collisions the system could detect false collisions. In this case the system can either return no solution, so it will take a new images and repeat all the process (the new segmentation could solve the problem), or find anyway a plan.

Comments In run 1 the robot grasps object o3 although there was o4 on top of it. This is because during the execution of (push_dir1 o0) action the gripper's cable touched object o4 moving it away. In runs 1, 4 and 5 the optimal plan does not return to push object o0 as a solution but it returns to push either o5 or o3, this also is because of the segmentation (at that iteration that system detected that it was not possible pushing away o0).

Due to the strange position of object o4 the grasping poses some times were the one depicted in Figure 7.5. When the planner returned to grasp o4 the inverse kinematic returned no solution, so it planed again performing a different action. The unfeasible grasping pose for object o4 was due to the height of the object which was higher than its width, so the second principal direction was perpendicular to the top largest surface (Figure 7.5). The situation was solved by taking new images, the Kinect was not able to see well the side of object o4 so sometimes the computed grasping pose could have solution. This can be easily solved by using more sophisticated grasping planning algorithms.

This thing about the grasping pose of o4 is worthy to mention?

Table 7.1: Executed plans of the 6 runs for the experiment of Figure 7.1. $N_{actions}$ is the total number of executed actions, $Time_{tot}$ is the total time to solve the task (neglecting the time lost due to the bad segmentation), $\neg Solution$ is the number of iterations in which no solution was found because of the segmentation and $\neg IK$ is the number of actions that had no solution for the inverse kinematic and the system replaned.

Run	Plan	$N_{actions}$	$Time_{tot}$	$\neg Solution$	$\neg IK$
1	(push_dir3 o2) (push_dir1 o1) (grasp o1) (push_dir1 o5) (grasp o5) (push_dir3 o2) (push_dir1 o0) (grasp o0) (grasp o3) (grasp o2) (grasp o4)	11	404.9s	3	6
2	(push_dir3 o2) (grasp o2) (push_dir2 o1) (grasp o1) (grasp o4) (push_dir2 o0) (grasp o0) (grasp o3) (grasp o5)	9	263.2s	19	0
3	(push_dir3 o2) (grasp o2) (grasp o4) (push_dir2 o1) (grasp o1) (push_dir1 o0) (grasp o0) (grasp o3) (grasp o5)	9	306.2s	3	1
4	(grasp o2) (grasp o4) (push_dir1 o3) (grasp o3) (push_dir1 o1) (grasp o1) (push_dir1 o5) (grasp o5) (grasp o0)	9	266.7s	0	0
5	(push_dir3 o2) (grasp o4) (push_dir1 o1) (grasp o1) (push_dir1 o5) (grasp o5) (push_dir1 o0) (grasp o0) (grasp o3) (grasp o2)	10	283.9s	0	0
6	(push_dir3 o2) (grasp o2) (push_dir1 o1) (push_dir1 o0) (grasp o4) (grasp o1) (grasp o3) (grasp o0) (grasp o5)	9	262.1s	6	0

Computation time analysis From the 6 runs of this experiment the different elapsed times for the several steps of the algorithm have been measured (Figure 7.6). The planning system is slowed down by the filtering process ($\approx$ 2.2*sec*), therefore other filtering algorithms should be considered. The segmentation is really fast since in average it takes half second to segment the image.

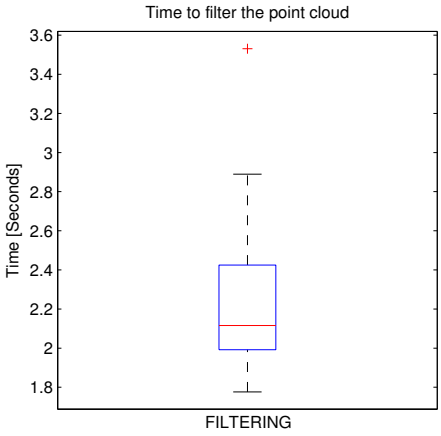
The most important part of this analysis regards the time needed to generate the states and get a plan (Figures 7.6c and 7.6d). The planning system is able to compute the predicates quite fast and, as commented in Chapter 4, its complexity is $\mathcal{O}(n^2)$. The time to get a plan depends on the complexity of the problem, that is on the number of objects. The Fast Downward planner showed to be very fast in resolving also complex problems with 7 objects. It is important also taking into account that our implementation to get the plan is not the optimal one since we write each time a PDDL file and then call the binary file of the planner. Hence, the process is slowed down by the writing of the PDDL file and the parsing of that file, and a better implementation should be considered. Despite this, the time to get a plan is few.

Regarding the time needed to compute the inverse kinematic is considerable and there is a big variance due to the fact that there exist poses for which is easier to find a solution and others for which is harder.

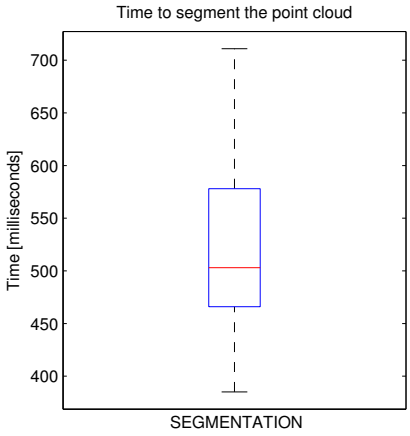
The execution of the actions is obviously the more expensive. The grasping actions takes more time than pushing since it is composed of more steps.

In Figure 7.7 the execution times per iteration of the perception and planning pipeline are depicted. We can observe that the whole system takes about $\approx$ 4.5*seconds* to take a decision after receiving a point cloud. The contribution of this thesis is at the level of planning and at the high level of the perception (i.e. the generation of the states), therefore the attention should be focused on the box regarding the state generation and the planning pipeline. Neglecting the filtering and the segmentation the system takes $\approx$ 2.3*seconds* to make a decision. Moreover, the IK depends on the robot and on the implementation, therefore we also showed the time performance neglecting the time to compute the inverse kinematic. It is possible observing that the system is fast since it takes in average 1 second, from the segmentation of the objects, to decide what action to do.

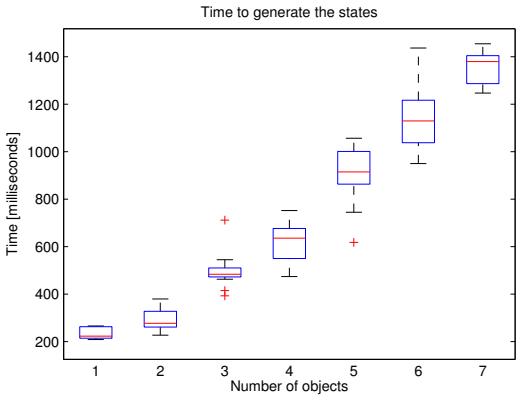
To conclude, we present in Figure 7.8 the elapsed times for the different phases of the proposed planning system for the 6-th run of the experiment. From that graph is appreciable that the worst part of the planning system is due to the low level perception (filtering and segmentation). The state generation takes some time for complex problems but for simple problems it is very efficient.



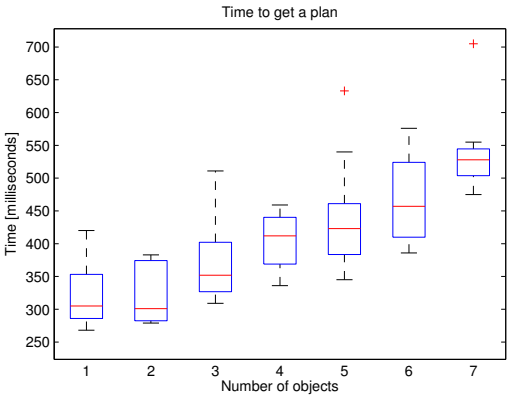
(a) Filtering time in seconds.



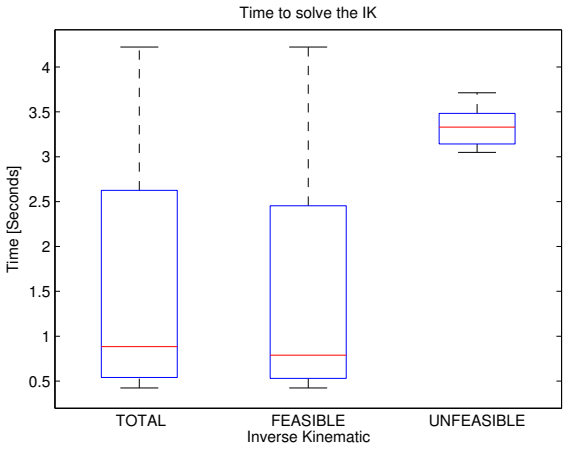
(b) Segmentation time in milliseconds.



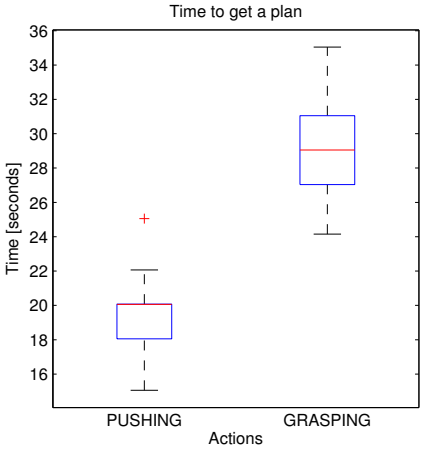
(c) State generation time in milliseconds.



(d) Planning time in milliseconds.



(e) Inverse kinematic time in seconds.



(f) Action execution time in seconds.

Figure 7.6: Elapsed time for the different phases of the algorithm. These data are taken from the 6 runs of the experiment of Figure 7.1.

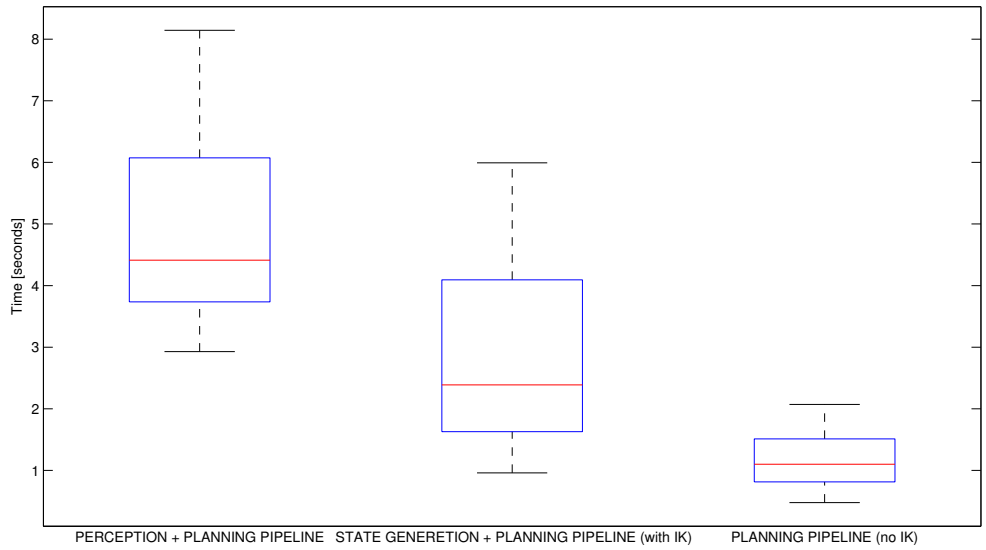
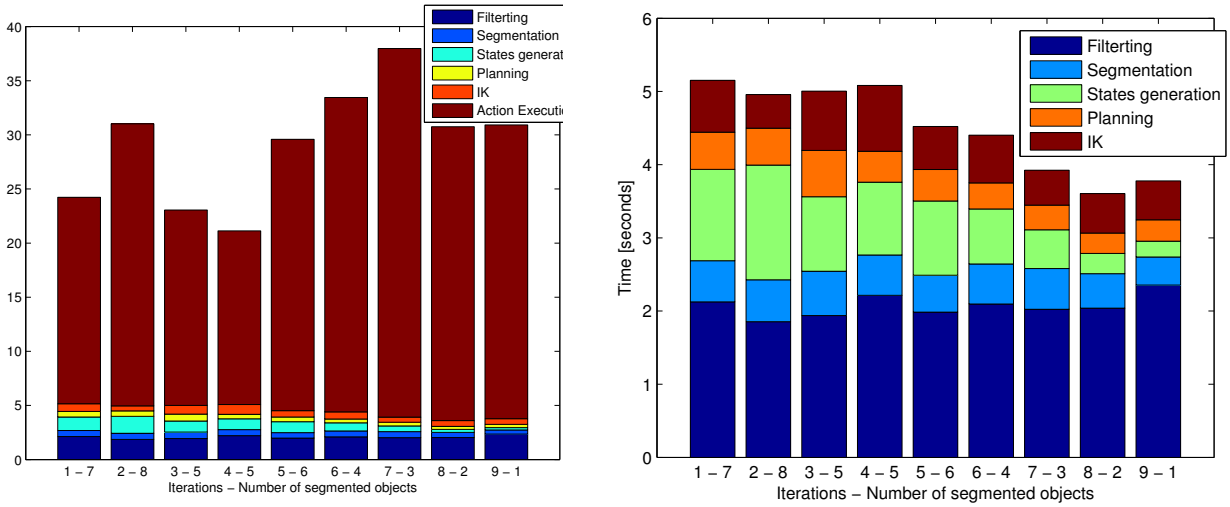


Figure 7.7: Time in seconds of the execution of the perception and planning pipelines. Those execution times are retrieved by all the runs without taking care about the number of segmented objects.



(a) **(b)**

Figure 7.8: Execution time for the different steps of the pipeline for the 6-th run of the experiment. Figure 7.8a includes the execution of the actions, while Figure 7.8b does not include the execution of the actions.

Chapter 8

Conclusions

8.1 Conclusions

8.2 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

8.3 Future Work