Master Thesis

Nicola Covallero

May 20, 2016

Contents

1. Abstract

2. Introduction

Robotic manipulation of objects is an increasing field of research which has struggled researches from many years ago. In several industries environments robots can be easily programmed considering some standard objects, i.e. the manipulation is always the same and known, and usually robot operations avoid to deal in cluttered scene, for instance through a conveyor belt the objects are singulated from the others ones. In order to move the robots outside industries and bring them to home of people they need to be equipped with a particular intelligence and skills that are still at a first stage. In this thesis we are going to investigate a simple approach that tries to replicate the human reasoning step during the manipulation of objects of table clearing task. The main objective is trying also to manipulate the objects in a human style, that is paying attention that during the manipulation the collision between objects is avoid as much as possible. This is a behaviour that all we do, of course with fragile objects, but also with objects that have no problem regarding collisions.

Next, in this chapter a brief introduction of the problem and the approach used is commented, and the experimental set up as well since it will be useful to understand some techniques and choice for the algorithm.

2.1. Problem Approach

In this section the approach to resolve the planning problem is described. All the strategy is inspired to a human like solution. A human to resolve such a task would use three main actions: grasp an object with one hand or both, when it is not possible to grasp an object because other object imped the human to put the hands in the proper way to grasp a desired object he/she has to interact with the objects in order to achieve a configuration of the objects that can be suitable to grasp an object, and this is achieved through pushing or dragging actions. Dragging an object is a very complex action which will need the robot to lay the end effector on the top of an object and make enough pressure on the contact in order that the friction between the table and the object is minor than the friction between the object and the robot's end effector. This is a very hard action which would imply an implementation of a reliable controller, because the goal is not only to move the object but also not to ruin it.

For these considerations the main actions the robot has to use in order to interact with the objects are:

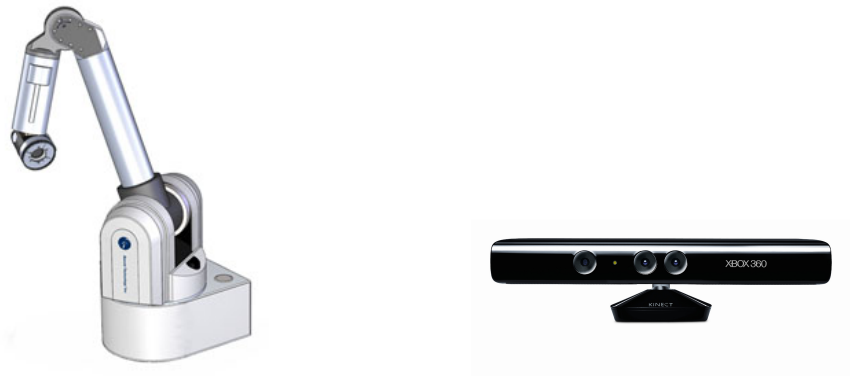
- Grasping
- Pushing

Grasping is the most important action since it lets to take an object from the pile of objects and drop it somewhere, for example in a bin, clearing in this way the table. There exist different works facing the same task by focusing only in grasping. The pushing becomes

useful considering the problem that two adjacent objects could not be grasped if they are so close such that the robot's gripper, when attempting to grasp an object is going to collide with an adjacent object, making such an object ungraspable. From this consideration is necessary the pushing action, in order to separate adjacent objects which could mutually exclude themselves to be grasped. Moreover the robot's intelligence is enhanced by a planning system which will return a sequence of actions in order to achieve the goal to clear the table.

2.2. Set Up

In order to make the understanding of the approach used the name the set up of the environment the robot will work in is presented. The robot used is a Barret WAM arm, which is a 7 degree of freedom (DoF) manipulator arm shown in Figure 2.1b.



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

Figure 2.1: Robot and vision sensor.

This robot is characterized by a low inertia of the end effector thanks to the kind of its actuators. The WAM is noteworthy because it does not use any gears for manipulating the joints, but rather a cable drive, 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 manipulate the objects the manipulation skills of the robot are enhanced by a gripper designed in the IRI institute driven by Dynamixel motors. Such a gripper is depicted in Figure 2.2 from several point of views.

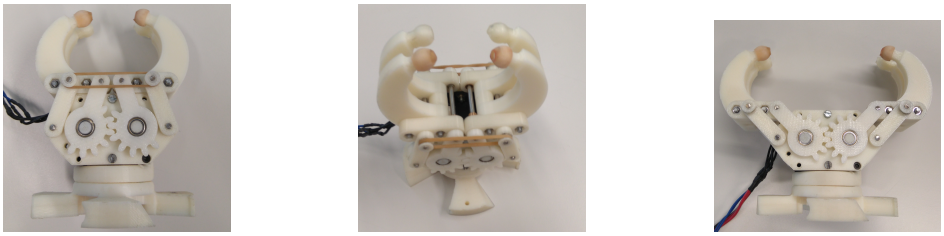


Figure 2.2: Gripper used for the experiments

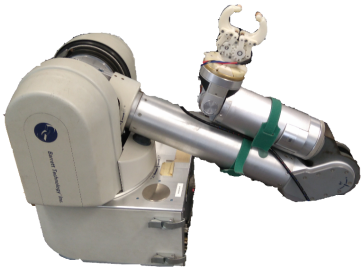
For the task planner, as the reader will see in chapters 4 and 5, the model of the gripper will be an important resource in order to find a feasible sequence of action that can free the table.

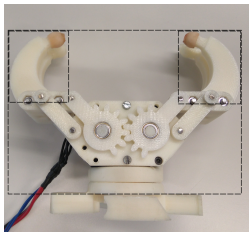
The gripper will be modelled measuring some principal elements such as: finger's width and height, gripper's width, height and deep, and the distance between the fingers when the gripper is opened or closed. The modelling procedure is depicted in Figure 2.4. The

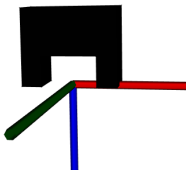
resulting model is a simple polygonal mesh which includes all the important geometric information of the gripper.

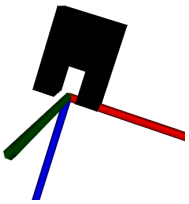
Such a simple model allows the collision algorithm commented in chapter 5 to check for collision in just few millisecond. A more detailed and complex model would only lead as benefits a higher precision, precision that in this case is not needed, and it will slow down algorithm because the collision checking procedure would be more computationally expensive. The gripper will be mounted in the end effector of the robot as shown in Figure 2.3.

Figure 2.3: Gripper and WAM.







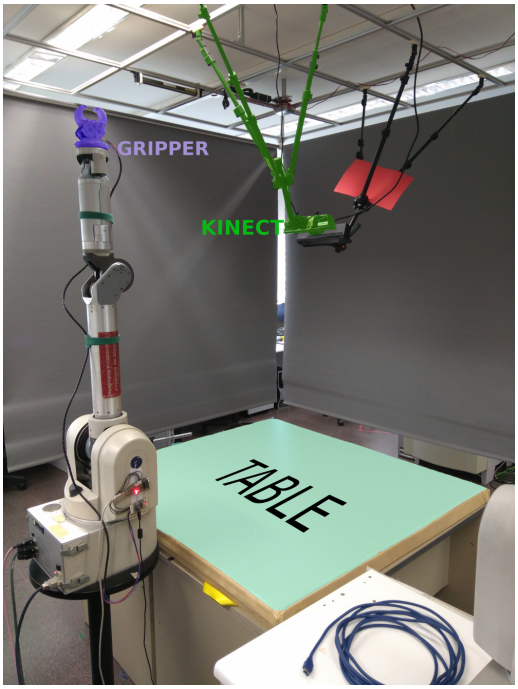


Elements measured Opened gripper mesh model Closed gripper mesh model

Figure 2.4: Modelling of the gripper: the fingers width, fingers height, the height of the whole gripper, its width and its deep have been measured. The red, green and blue axis are respectively the x, y and z axis.

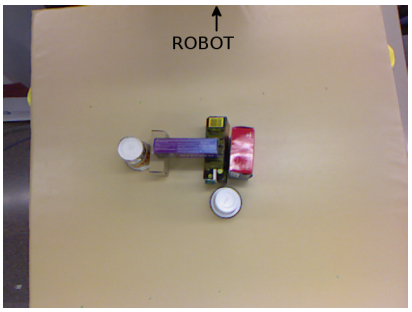
The scenario the robot is going to work in is presented by a table, and the objects will lay on top of it. In Figure 2.5a the main elements of are highlighted. The WAM arm is in a fixed position with respect the table and the Kinect camera is located on top of the table pointing down. The Kinect is calibrated 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 frame. An example of a cluttered scene the robot is going to deal with is depcited in Figure 2.5b, and the same scene seen by the Kinect is shown in Figure 2.5c. With respect the kinect's view the robot is located at the top of the image, outside the point of view of the Kinect in order to avoid to detect the arm as an object. If the robot arm would be present in a depth image we should apply some algorithms to identify the robot arm, which could also produce important occlusions, that is hide some objects from the Kinect's view. In order to avoid all this, the depth image considered is the last one provided by the Kinect when the robot is no more in the Kinect's view (these poses are known a priori).

Concerning the example of Figure 2.5b it can be appreciated that the sequence of operation needed in order to clean all the table with a careful manipulation is quite complex, and due to the gripper geometry, and to the grasping poses we choose (section **??**), the robot will need to grasp first the purple top object, or the white cup, then move one box in other to grasp the one of its side. All this sequence of actions will be decided by the robot.



(a) Principal elements of the experimental set up.

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



(c) Kinect's view.

Figure 2.5: Experimental set up, and example of a cluttered scene the robot is going to interact with.

3. State of the Art Manipulation Planning

cite David's works

still to add reference to the ones in the pushing.odt file

Many manipulation planning approaches (see [30] for an overview) assume that the task can be treated as a geometric problem, with the goal to place the objects in their desired positions. Planning is essentially done with a mixture of symbolic and geometric states, this requires a set of symbolic predicates that correspond to geometric relationships. In particular to the best of my knowledge, all the planners for cluttered scene are based on a mixture of symbolic and geometric predicates, or based only on geometric ones.

Dogar and Srinivasa [19] proposed a framework for planning in clustter scenes using a library of actions inspired by human strategies. They designed a new planner that decides which objects to move, the order to move them, where to move them, and it chooses the manipulation actions to use on these objects, and accounts for the uncertainty in the environment all through this process. The planner first attempts to grasp the goal object, than if it is not possible it identifies what is the object that obstacles the action, then such an object is added to a list of the objects that have to be moved. They moved an object in whatever position such that makes the goal feasible. Despite this, such a planning strategy can be used only in the design of a new planner, which is something we want to avoid.

To grasp they used the Push-grasping action [18] , which is a robust way of grasping objects under uncertainty. It is a straight motion of the hand parallel to the pushing surface along a certain direction, followed by closing the fingers. For the pushing directions they used a resolution of $\pi/18$ rad (i.e. 36 different directions) and use a predefined set of 9 hand aperture values. However their cluttered scenes is intended to be a scene with separated and well known objects.

In [23] the authors proposed to apply a linear temporal logic (LTL) planner to a manipulation planning task in cluttered scenes, but it suffers from poor runtime and the LTL specification is complex. Moreover to the scope of this work the temporal specification is not needed since it is indirectly encoded in the clutter scene composition.

An interesting planner which mixes symbolic and geometric predicates is aSyMov [13]. It is probably the most well known planner which a task to move objects considering how their location will then affect the possible robot's paths. To do that they combine a symbolic planner with a probabilistic roadmap [41] for geometric planning.

A recent alternative proposed by Mösenlechner and Beetz [35] is to specify goals symbolically but evaluate the plan geometrically. The idea is to use a high-fidelity physics simulation to predict the effects of actions and a hand-built mapping from geometric to symbolic states. Planning is conducted by a forward search, and the effects of actions determined by simulating them, and then using the mapping to update the symbolic state. This could be look promising to the scope of this thesis but still implies the design of a new

planner.

In [32], Lonzano-Peréz et al. They use a simple task-level planner, in which operators are described with two types of preconditions: symbolic and geometric. They proposed a strategy for integrated task and motion planning based on performing a symbolic search for a sequence of high-level operations, such as pick, move and place, while postponing geometric decisions, based on the CSP (Constraint Satisfaction Problem) technique. **Their technique allow working directly with state of the art planners.**

In [14] the authors address a problem similar to the one of this thesis. Pushing presents itself as a more intricate problem than grasping, a for grasping, after the object has been grabbed, it becomes a matter of solving a geometric problem. However, pushing an object carries uncertainty to the resulting state of the object. It is, nonetheless, important to be able to incorporate these two actions in the planning. For pushing they used a similar strategy to [34]. The planning algorithm also incorporates grasping, which, together with push actions, can perform a vaster array of tasks. It uses the concept of **reachability** [44] for each kind of action to quickly exclude impossible poses of the gripper of the planning stage, creating a reliable plan suitable for real-time operation. The authors model the planning problem through MDP (Markov Decision Process), discretezing the world in grid cells and assigning to each one a push and grasp vector definied by the reachability concept. Moreover each object is also classified through a simple primitive shape in order to have a proper map of pushing and grasping for each kind of considered object.

The majority of the state of the art for task planning in cluttered scene is focused on designing a new planner, why in this thesis state of the art planners ready to use want to be used.

Learning in Task Planning Machine learning techniques also have been applied to task planning. To plan complex manipulation tasks, robots need to reason on a high level. Symbolic planning, however, requires knowledge about the preconditions and effects of the individual actions. In [9] the authors proposed a practical approach to learn manipulation skills, including preconditions and effects, based on teacher demonstrations. It is difficult to solve most real world manipulation tasks by reasoning purely in terms of low-level motions due to the high-dimensionality of the problems. Instead, robots should reason on a symbolic level and appropriately chain the learned actions to solve new tasks. Such a planning step, however, requires knowledge of the important preconditions and effects of the actions. With few demonstration the authors proposed a method to teach the robot the precondition and effects of individual actions. Their method furthermore enables the robot to combine the learned actions by means of planning to solve new tasks that are more complicated than the learned individual actions. They focused their work on a class of manipulation actions where the preconditions and goals depend on spatial or geometrical constraints between the involved objects. They describe a scene as a collection of features and during the action demonstration the algorithm looks for the features that have a small change to identify the ones that can be predicates or effects of the considered action. They use an heuristic metric to define the variation of the features. The problem of this approach is that it moves the problem of identifying the right predicates to describe the problem in identifying right features that let a correct learning of the predicates. **We could consider this as further work**.

In [17] [16] the authors propose an approach for planning robotic manipulation tasks which uses a learned mapping between geometric states and logical predicates. The planner first plans symbolically, then applies the mapping to generate geometric positions that are then sent to a path planner. They try to fit a probability distribution function to the geometric states to map them to a symbolic state. This work sounds promising but the

cluttered scene this thesis is going to treat is quite complex to be treated with such a method.

Interpreting the scene Focusing on symbolic planning, 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 [20] [36] [21] the authors propose a system which combines 3D recognition and segmentation results to create and maintain a consistent world model involving attributes of the objects and spatial relations among them. Unknown objects are modelled by using the segmentation output to determine their sizes and considering similarities with existing models to determine their shapes and colors. Then, these models are also stored as templates to be used for recognition along with the extracted attributes. They focused on the extraction of size, shape and color attributes as well as the following unary and binary spatial relations: on, on ground/on table, clear and near for object manipulation scenarios. These predicates are generated and updated over time to maintain a consistent world model. **For the aim of this thesis this paper is of particular interest because let to compute nicely some useful symbolic predicates** The *on* relation for a pair of objects is determined by checking whether their projections on the table plane overlap. For each pair of objects the *near* relation is determined by comparing the distance between the centers of these objects in each dimension with the sum of the corresponding sizes in that dimension.

3.1. Usefull things

Temporal filtering to reduce the noise of the kinect [36].

3.2. Usefull Concepts

Closed world assumption [39] can be defined as having complete knowledge about the world, that is, the numbers and the attributes of all objects are known apriori.

4. Task Planner

In this chapter the general framework adopted is discussed, proposing a suitable task planner. 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.

4.1. State of the Art

As already seen in chapter 3 there exist different kind of planners and they can be grouped in three main categories:

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

Classical planners, as suggested by the name, are the more classical and easy to use. They 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..) [40]. A deterministic problem is generally formulated as a 6-tuple $\Pi = \langle S^d, s_o^d, G^d, A^d, T^d, c^d \rangle$ [31], where:

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

The solutions, or trajectorys, 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 $\sum_{a \in \tau} c^d(a)$. The optimal solution is the solution with less cost: $\tau^* = \min_{\tau_i} c^d(\tau_i)$. A very well known classic planner is the Fast Downward planner [24].

Hierarchical planning, also called *Hierarchical Task Network*(HTN), works in a similar way to how it is believed that human planning works [33]. 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. In order to tell the planner how to decompose nonprimitive tasks into subtasks, it 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 [1]. For this kind of planning technique a well known planner is JSHOP2 [4]. **Probabilistic planning** is a planning technique which consider that the environment is not deterministic but probabilistic. So the actions have a certain probability to obtain a certain state, and given an initial and final state the planner finds the solution path with the highest probability. A well known example on which this kind of planners are build on is the Markov Decision Process. A probabilistic problem is generally formulated as a 6-tuple $\Pi = \langle S, s_o, G, O, T, A \rangle$ [31], where:

- S is a finite set of states;
- $s_o \in S$ is the initial state;
- $G \in S$ is a goal state;
- O is the set of outcomes, the probability of $o \in O$ is $Pr(o)$;
- $T(o, s) \in S$ is a (total) deterministic transition function for all outcomes $o \in O$ and states $s \in S$;
- $A(s)$ is a set of applicable actions for each $s \in S$, coupled to a function $out(a) \subseteq O$ mapping each action to a set of outcomes in such a way that
  - each outcome $o \in O$ belongs exactly to one action $act(o)$;
  - $\sum_{o \in out(a)} Pr(o) = 1$ for all a.

In this case the optimal solution is the one with the highest probability. In this category two famous probabilistic planners are Gourmand [2] and PROST [3].

4.2. Planner

The problem this thesis is facing could be resolved by several approaches by using planners from all the categories. We have already seen that such a problem involves geometric constraints and those cannot be considered directly by the planner using a ready to use state of the art planner, that would imply a designing of a new planner. Since the aim of this work is not to design a new planner but to resolve the table clearing task through already existing planners the problem has to be cast in a way to be manipulated by existing planners. This easy way involves working with symbolic predicates, symbolic predicates are predicates which can be true or false, and they will be introduced more in detail in the next sections.

The problem moreover involves a big amount of uncertainty due to the interaction of the robot with the environment. When the robot will interact with the pile of objects it is very hard to predict correctly the position of the object in the next frame, that is the next state, this is a crucial problem which should be considered. With a probabilistic planner, the planner will take into account what object has been moved and it will update the state obtaining a set of states, each one with a certain probability, and the returned plan, or solution, is the one with the highest probability. The probability also has to be modelled and it is highly depending on the form of the object, which is also hard to predict. An other way to face the problem is to replan at each frame, that is after each interaction of the robot with the pile of objects, or whenever the current state deviates from the expected one, generating a new trajectory from the current state to the goal. The plan's actions are considered deterministic, and the only useful action is actually the first one of the plan, after its execution the system replans again. Little et al. discussed in [31] the problem of when is more useful the probabilistic planning with respect a simple replanning system. They defined a the concept of *Probabilistic Interesting Problem* with the following definition:

A probabilistic planning problem is considered to be *probabilistically interesting* if and only if it has all of the following structural properties:

- there are multiple goal trajectories;
- there is at least one pair of distinct goal trajectories, τ and τ', that share a common sequence of outcomes for the first $n-1$ outcomes, and where τ_n and τ'_n are distinct outcomes of the same action; and
- there exist two distinct goal trajectories τ and τ' and outcomes $o \in \tau$ and $o' \in \tau'$ of two distinct actions $a = act(o)$ and $a' = act(o')$ such that executing a strictly decreases the maximum probability of reaching a state where a can be executed.

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

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

Taking into account such considerations, in order to face simply such a difficult problem, the problem has been thought to be solved thanks a deterministic replanner although it is *probabilistic interesting*. The choice also was guided by the difficulty to model the probability distribution of the actions which depends on the particular shape of the objects the robot has to interact with.

Between a classic planner and an heuristic one there is not much difference for the aim of this work. The planner chosen to develop the work is the **Fast Downward** planner [24][5], a very well know classic one. The advantage of this planner is its wide documentation with respect other planners, which makes it easier to use and it also has a wide community.

4.3. Symbolic Predicates

One the planner has been chosen, a task planning problem has to be formulated accordingly to the capabilities of such a planner. The *Fast Downward* planner needs the problem to be formulated in the common format of PDDL (*Problem Domain Description Language*) [7].In this section the symbolic predicates that have been considered in order to resolve the problem are described.

The task this thesis is going to resolve is a common task done by humans who thinks in order to find a feasible sequence of actions. Such a sequence is normally composed of actions that avoid the collision between the manipulated object and the other ones, whenever possible. To do this we, as human, think on what is going to happen if we manipulate an object in a certain way, and what objects can be manipulated at the current instant and what objects need to be moved in order to interact with a certain one.

The design of the problem has been inspired on such a reasoning way. The reasoning part is entrusted to the planner, which needs a coherent problem description through symbolic predicates. How 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 in a bin, while the pushing action is an auxiliary action which has the aim to move an object in order to collocate it in a pose that do not obstacle the solution of the plan. In fact it happens that an object cannot be grasped because there is an adjacent one that impedes such an action.

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

- When cannot an object be grasped?
- When can a object be pushed? In which direction?

Grasping Action An object cannot be grasped always but only if the grasping action is not imped by other objects, to do this the following things have to be checked:

- no objects have to stand on top of it
- there are no adjacent objects that imped the grasping action

We don't want to grasp an object which has another one on top of it for one main reason: during the grasping action such object will fall corrupting the scene and such behaviour is undesired, in fact an human will never grab an object at the bottom of another one without first grabbing the one on the top. To include such an information in the problem description the predicate **on** is created, in particular it is defined as `(on o1 o2)` meaning that object labelled as `o1` stands on top of `o2`.

Once we are sure the current object has not objects on top of it we have to check if it can be grasped, that is, given a grasping pose we have to check if that pose is feasible or if it is imped by other objects. To include such an information in the problem description the predicate **block_grasp** is created. In particular it is defined as `(block_grasp o1 o2)` meaning that object labelled as `o1` impedes object `o2` to be grasped.

Pushing Action In certain situation when the grasping is not possible because an object impedes another one to be grasped one of both has to be moved. Firstly, being observant of the philosophy to avoid manipulation that can damage the objects, we considered that only the objects that stands on the table can be pushed, that is an object on top of another one will be no pushed. And, how discussed for the grasping action, also an object that has on top of itself another one will be no pushed. Second the possible directions along with an object can be moved have to be decided. Such directions are taken accordingly to the shape of the object, in particular to its principal directions. Moreover the pushing direction is considered to be parallel to the table plane, as a normal human would be. Therefore the principal direction is projected onto the table plane, and also the orthogonal direction to the principal one is considered, having in total 4 possible ways to push an object, 2 ways per direction. To the best of my knowledge there are no studies about how pushing an object on a proper way, or direction, in order to make it follow a desired trajectory.

Having the direction each object can be pushed along with, the next step is to understand if the object can be pushed along those directions. An object cannot be pushed in a certain direction if there is another one that blocks the current object. For each direction the object is therefore translated in the pushing direction and checked for collision with other objects. In this way it is possible to get what is the object which impedes the considered one to be moved along the considered direction.

To include such an information the predicates **block_dir1**, **block_dir2**, **block_dir3** and **block_dir4** are created. In particular, they are defined as `(block_dir`$_i$ `o1 o2)` meaning that the object labelled as `o1` is impeding object `o2` to be moved along direction `dir`$_i$.

Effects of the action The effect of the action has to change the current predicates of the problem in order to update coherently the state. For the *grasping* action whenever an object is grasped, since we are working in a deterministic scenario, the object is supposed

to be grasped successfully and drop in the bin. If such an object was on top of other ones or it was impeding some objects to be pushed in some direction those predicates have to be update coherently. Moreover to considered that the object is no more in the scene the predicate **grasped** is added. In particular `(grasped o1)` means that the object labelled as `o1` has been grasped and dropped into the bin, and therefore there is nothing more to do with it. The *pushing* action, still thinking in a deterministic scenario, is supposed to be done in a manner that the object will be moved far enough from the others in such a way that it can be considered isolated. Considering it is isolated from the others, if it was blocking some object to be pushed in a certain direction now it will not impedes no more, and the same if an object impeded the current one, so the $\texttt{block\_dir}_i$ predicates are update, and the `block_grasp` as well.

Geometric constraints The proposed planner until now does not include any geometric information, such as whether the robot can actually perform a certain action. It may happens that the object is outside the working space of the robot and this makes it hard to interact with. In such a case the robot will have a plan but it is impossible its execution, this make the plan unfeasible.

Add concept of hybrid planning

This aspect has to be taken into account and it can be done using a common technique in task planning such as **backtracking** [11]. The backtracking technique is about compensating the gap between planning with abstract actions and some physical and geometric constraints such as kinematics. Backtracking is based on updating the initial state used to get the current plan considering the geometric and kinematics constraints, and replan again with the updated initial state, and obtaining in this way a different plan that could be feasible. The obtained plan will still be evaluated and backtracked until a feasible plan is obtained or there exists no plan. The pseudo algorithm is shown in Algorithm 1.

Algorithm 1 Planning procedure

procedure GetPlan($state_{init}$,$goal$) ▷ Initial state and goal
  repeat
    $plan$ = GetFastDownwardPlan($state_{init}$,$goal$)
    if isPlanEmpty() **then return** NULL
    end if
    $action, object$ = GetFirstAction($plan$) ▷ Action to execute and object of interest
    $success$ = HasIKSolution($action, object$)
    if $\neg success$ **then**
      $state_{init}$ = UpdateInitialState($action, object$) ▷ Update $state_{init}$ adding a predicate that avoid to execute $action$ on $object$
    end if
  until $success$ **return** $plan$ ▷ Return a geometric feasible plan or no plan at all
end procedure

This technique is a very common technique for planners that plans symbolically and have no informations about geometric constraints. This is considerable limitation of symbolic planning but makes it very fast to execute.

Let's consider to include the geometric information, such as path planning or inverse kinematics, into the geometric planner. The inverse kinematics by itself is an expensive operation, overall considering the high degree of freedom of the WAM arm. The most of the times the objects are considered to lie inside the working space of the robot. Computing

the inverse of Kinematics for each possible action for each object would be very expensive, and by experimental result the majority of the times the inverse kinematics is feasible and the majority of the plans are feasible. Moreover, due to the probability of the next states the planner has to replan at each frame, and computing the predicates means computing the inverse kinematics as well. All this would make the planner process extremely time consuming when it is not needed.

To consider the geometric unfeasibility of an action a predicate for each action is added: `ik_unfeasible_dir1 ?o`, `ik_unfeasible_dir2 ?o`, `ik_unfeasible_dir3 ?o`, `ik_unfeasible_dir4 ?o` and `ik_unfeasible_grasp ?o`. The backtracking procedure will try to execute the action by first computing a solution for the inverse kinematic for the path to execute, the pushing action will be a sequence of points in the Cartesian space and the grasping action will be composed of a pre-grasping pose and a grasping pose, and then if that action on a particular object is not possible to execute one of those predicates is added.

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

Include this in the experiments?

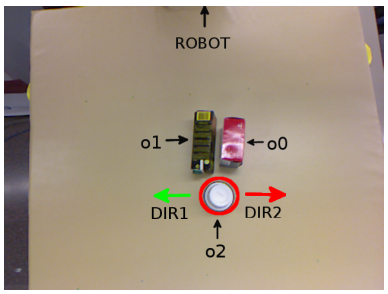


Figure 4.1: Unfeasible plan due to geometric constraints. In this case the planner returns as action to execute grasping or pushing away the white cup (highlighted by a red circle) but it is out the configuration space of the robot and there exist no plan for that problem. This problem is the one depicted in Figure 2.5b, where the other objects has been grasped as consequence of the plan solution.

A situation in which the backtracking is useful is shown in Figure 4.1. Accordingly to our strategy, the robot cannot be grasp the black or red box because the gripper would collide, and the same for the pushing action. It has to interact with the white cup in order to make space to move the other objects and grasp all of them. The planner first returns the following plan: `grasp o2`, `push_dir1 o0`, `grasp o1`, `grasp o0`. It will try to solve the inverse kinematic for the `grasp o2` action, but it found no solutions. Therefore the predicate `ik_unfeasible_grasp o2` is added and the planner replans. The new plan is: `push_dir1 o2`, `grasp o2`, `push_dir1 o0`, `grasp o1`, `grasp o0`. In this case it will try to get a solution for the inverse kinematic for the pushing action but it finds no solution, so the predicate `ik_unfeasible_dir1 o2` is added. And so on until the planner find that all

the possible actions for a feasible plan have no solutions for the inverse kinematic. In this way the planner includes the ability to understand also why there exist no plan for a certain problem.

Add the case in which the robot cannot grasp an object but it can push in a pose easier to grasp?

Add pipeline Scheme

It may happen that the object outside the working space of the robot blocks the execution of the task because if the planner insists to return as first action an action with that object, then it will find that is not possible to interact with it and therefore there exist no solution on the planner procedure. To overcome this, problem, and make the problematic objects left as the last one to interact with if possible, the actions are enhanced with another effect, that is all the objects that have one of the unfeasible inverse kinematic predicates set to true, are set to false. In this way, if is possible, the only the first action consider the geometric constraints, while the next ones have no information about the geometric constraints.

ADD EFFECT

PDDL For clarity purposes, the PDDL syntax of the described action is here reported. For the *grasping* action its PDDL syntax is shown in listing 4.1.

```
(:action grasp
    :parameters (?o - obj)
    :precondition (and
                  ; grasp it if there are no objects on its top
                  (not (exists (?x - obj)(on ?x ?o)))
                  ; grasp it if there is no object that blocks it
                  ; to be grasped
                  (not (exists (?x - obj)(block_grasp ?x ?o))))
    :effect (and
            ; the object "o" is grasped
            (grasped ?o)
            ; if the object was on top of other ones now it is
            ; no more on top of them
            (forall (?x - obj)
              (when (on ?o ?x) (not (on ?o ?x))))
              ; the grasped objects no more blocks other objects
            ; to be pushed or grasped
            (forall (?x - obj)
              (when (block_grasp ?o ?x) (not (block_grasp ?o ?x))))
            (forall (?x - obj)
              (and
              (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))))
              ; remove all geometric constraints
              (when (ik_unfeasible_dir1 ?x)(not (ik_unfeasible_dir1 ?x)))
              (when (ik_unfeasible_dir2 ?x)(not (ik_unfeasible_dir2 ?x)))
              (when (ik_unfeasible_dir3 ?x)(not (ik_unfeasible_dir3 ?x)))
              (when (ik_unfeasible_dir4 ?x)(not (ik_unfeasible_dir4 ?x)))
              (when (ik_unfeasible_grasp ?x)(not (ik_unfeasible_grasp ?x))))))
```

Listing 4.1: PDDL syntax of the grasping action

For the *pushing* action its PDDL syntax is shown in listing 4.2.

```
(:action push_dir1
    :parameters (?o - obj)
    :precondition (and
```

```
                    ; 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))))
                ; remove all geometric constraints
                (when (ik_unfeasible_dir1 ?x)(not (ik_unfeasible_dir1 ?x)))
                (when (ik_unfeasible_dir2 ?x)(not (ik_unfeasible_dir2 ?x)))
                (when (ik_unfeasible_dir3 ?x)(not (ik_unfeasible_dir3 ?x)))
                (when (ik_unfeasible_dir4 ?x)(not (ik_unfeasible_dir4 ?x)))
                (when (ik_unfeasible_grasp ?x)(not (ik_unfeasible_grasp ?x)))))
```

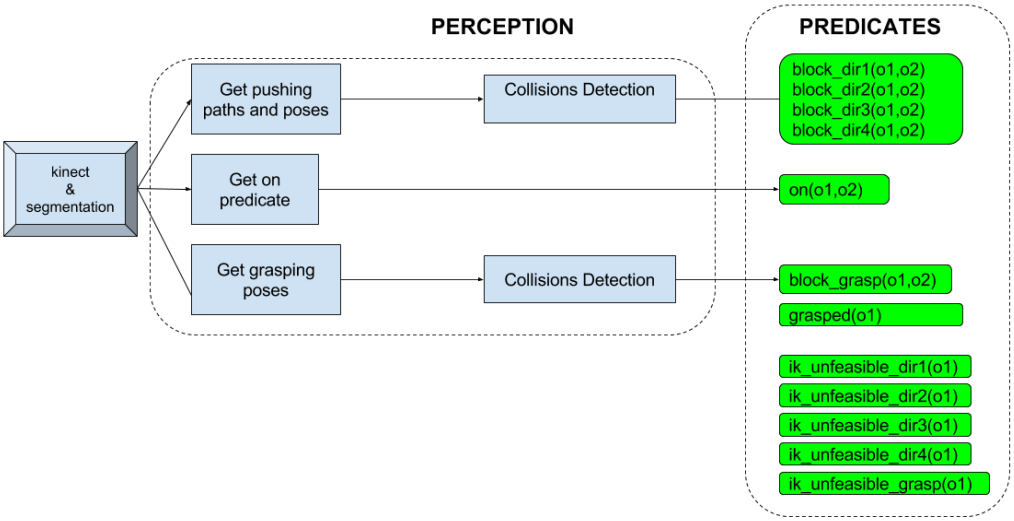
Listing 4.2: PDDL syntax of the pushing action along direction 1

With such predicates the planner has all the basic information about the location of the objects on the scene. The symbolic predicates try to simplify the problem translating the geometric properties of the objects in symbolic ones. 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 more uncertainty is related to the pushing action, the direction selection, based on the principal axis, does not taken into account reliably the geometry of the object. The trajectory will be unlikely the desired one but a similar one. In the case the result of the action is not the expected one the replanning strategy will resolve the uncertainty problem. In this manner we are forced to considered that some interactions between objects are allowed but tried to be avoid as much as possible.

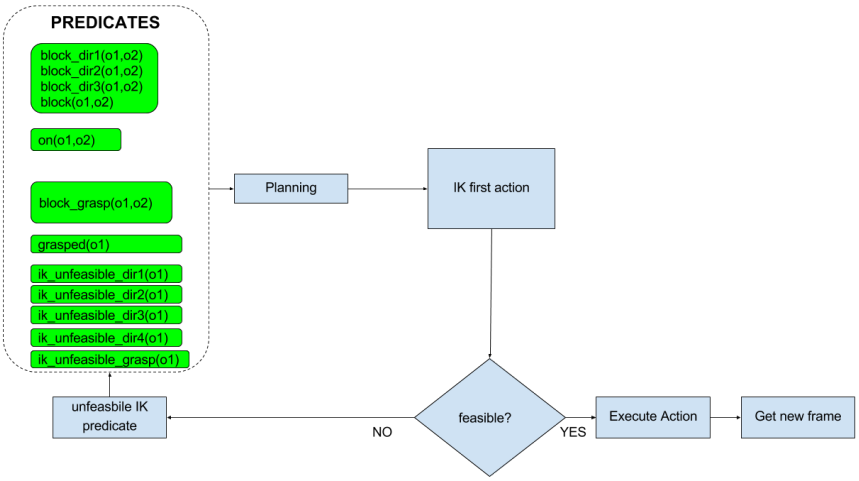
In order to provide a graphical scheme to the reader the planner pipeline is depicted in Figure 4.2. It can be appreciated the two fold strategy of the algorithm, the first stage (Figure 4.2a) is devoted to get the predicates from the Kinect sensor and the second one to evaluate the feasibility of the returned planned through backtracking and to execute the plan(Figure 4.2b).

Notice that the `grasped` predicate is just a predicate used only by the planner in order to know when the object has reached the goal or not, it does not need to be updated by the task planner.

The way of computing the predicates will be discussed in detail in the next chapter.



(a) Perception Pipeline



(b) Planning Pipeline

Figure 4.2: Planner pipeline

5. Algorithm

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

5.1. Object Localization

For the planner, to know how to move the object, is fundamental knowing the objects on the scenario, therefore they need to be detected. More correctly, they need to be segmented since the algorithm is dealing with unknown objects, it is not going to recognize an objects as a particular one, but it segments the objects.

This stage is composed on two steps:

1. Detecting the table top objects
2. Segmenting the table top objects

The depth sensor is recording a depth map of a table, the algorithm has therefore to detect first the table, and so the objects that stands on top of it, and then segmenting them. We don't want to segment the entire image, if so, the table will be segmented as an object and the floor as well.

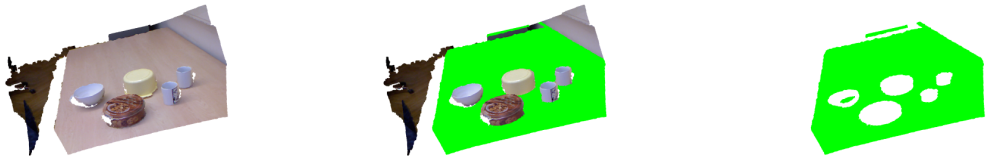
5.1.1. Tabletop Object Detection

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

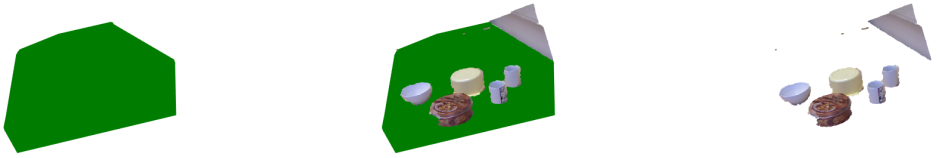
1. **Table plane estimation** (by RANSAC): the points of the table are detected estimating first a plane in the point cloud, all the points which belong to such a plane are the points of the table.
2. **2D Convex Hull of the table**: having the points of the table a 2D convex hull is computed in order 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 steps of this tabletop object detection algorithm are described in Figure 5.1 for the point cloud[1] in Figure 5.1a.

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



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



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

Figure 5.1: 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).

5.1.2. Object Segmentation

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

Supervoxel

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

In this work the supervoxels are computed with the Voxel Cloud Connectivy Segmentation (`VCCS`) algorithm [38], which was proposed in 2014 and gives good improvements with respect to previous state of the art methods, and still it is able to be used in online applications.

The algorithm works in 4 main steps:

- Voxelizing the point cloud
- Creating an adjacency graph for the voxel-cloud
- Creating seeds for the initial supervoxels centres
- Each seed is described by 39 features that describe spatial coordinates, colors and local surface model properties. Then a distance metric based on these features is defined.
- Clustering the voxels into supervoxles iteratively by means of the distance metric, the adjacency graph, and the search volume of the supervoxel.
- Once the search of all supervoxel adjacency graphs has been concluded, the centres of each supervoxel is updated by taking the mean of all its constituents.

Local Convex Connected Patches Segmentation

Once the supervoxels of the 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`) [42]. This algorithm permits to segment objects by clustering together adjacent convex supervoxels. The algorithm is quite simple but very good for segmentation of objects that have convex shapes, in Figure 5.2 the algorithm is briefly described.

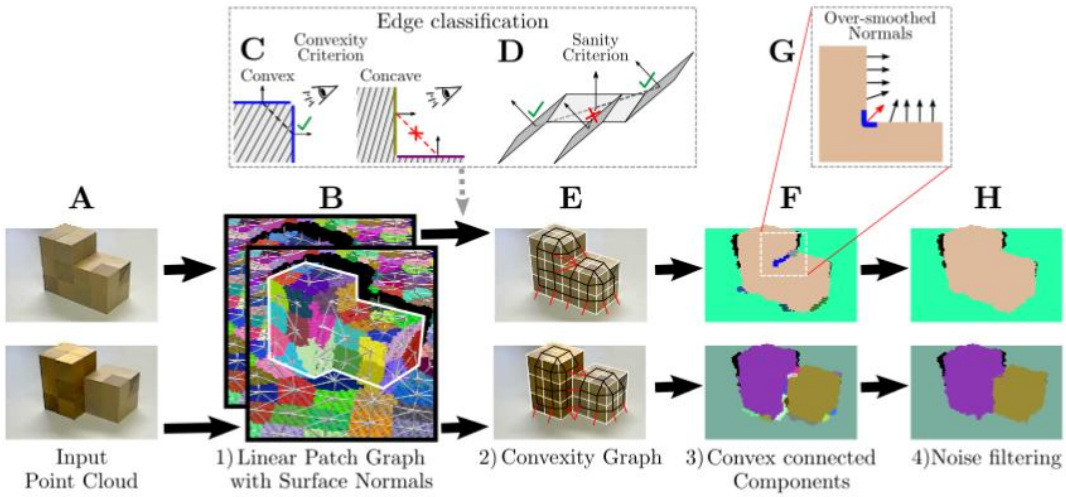


Figure 5.2: `LCCP` algorithm's structure. Reproduced from [42]

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

- Extended criterion: to consider two adjacent patches convex, both must have a connection to a patch which is convex with respect both patches
- Sanity Criterion: check if the adjacent patches which can be considered as convex present geometric discontinuities (see point D of Figure 5.2), 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 5.2), 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 5.3.

Figure 5.3: Example of segmentation results.

Note that the algorithm works mainly 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 no only one color.

A color based segmentation could segment a draw, 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 5.4 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. Vice versa, any interaction with the box is impossible without having a collision with the green stripe. This is the main reason the segmentation we used is based only on geometric features (The `LCCP` algorithm has the option to include color importance but we neglected it).

Figure 5.4: Box with a green stripe.

Talk about filtering

5.2. Background

In this section will be presented some concepts that will be used to execute the actions and to compute the predicates.

Principal Components The first concept to know is the concept of principal directions. 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.

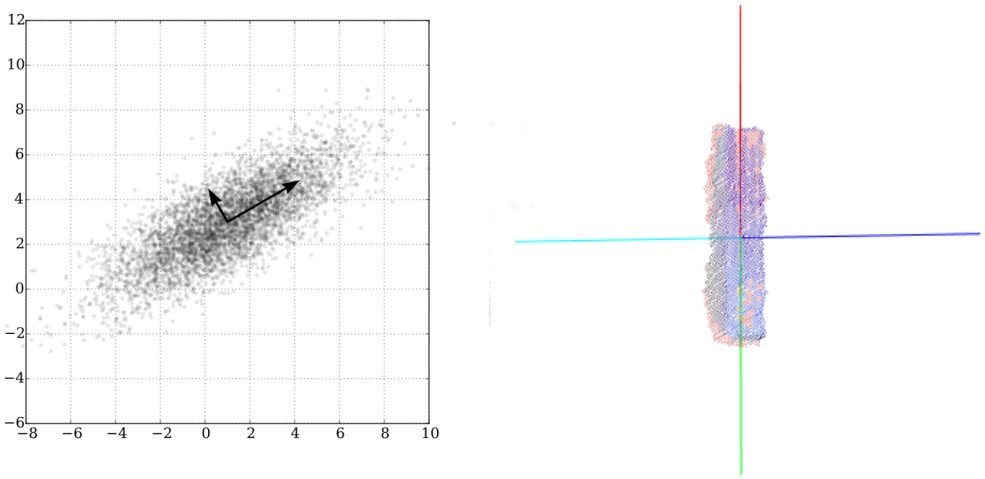
To obtain the principal axis the principal component analysis (PCA) [26] technique is used. This technique is a common statistical procedure that uses orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, which 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 5.5a[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 in to the object's point cloud to retrieve its principal components, or how called in this work principal directions. In Figure 5.5b the principal directions of a generic object are illustrated.

Projection onto a plane We will see later that several things need the concept of the projections of a point into a plane. Considering a point $p = x_p, y_p, z_p$ and a plane P defined by the following equation

$$ax + by + cz + d = 0 \tag{5.1}$$

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



(a) **(b)**

Figure 5.5: Principal Components Analysis - In Figure 5.5a PCA for a standard 2D set of observations. In Figure 5.5b results of the PCA for a rectangular segmented object. The green, red lines refers to different ways of the first principal components, while blue and cyan lines refers to different ways of the second principal directions. The third one would be orthogonal to the first two principal components.

the projection p_P of point p onto the plane P is given by the following set of operations:

1. Given the origin point $P_0 = (x_0, y_0, z_0)$ of the plane, which can be calculated by arbitrary x_0 and y_0 coordinates as

$$z_0 = \frac{-1}{c}(ax_0 + by_0 + d),$$

2. calculate the coordinates of P_0 with respect the point p

$$P^p = p - P_0,$$

3. then calculate its projection λ_p onto the plane normal $n = (a, b, c)$

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

4. compute the coordinates of the projection onto the plane of point p with respect to the point itself:

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

5. translate it by the coordinate of the points in order to have the projection of the point expressed with respect the original reference frame:

$$p_P = \bar{p_P}^p + p.$$

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

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

$$R_B^A = \begin{bmatrix} {}^A\hat{X}_B \\ {}^A\hat{Y}_B \\ {}^A\hat{Z}_B \end{bmatrix}$$

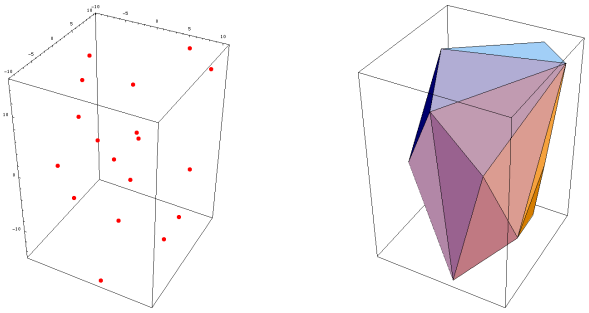
To transform any object, such as the gripper mesh model, to frame $\{B\}$ then the following homogeneous transform is applied:

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

where $R_A^B = R_B^{A\top}$ and ${}^A B_O$ is the origin of frame $\{B\}$ relative to $\{A\}$. In this way, having some axis that defined 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 the one defined by the new reference frame.

Axis Aligned Bounding Box (AABB) Axis aligned bounding box is a bounding volume defined with respect the reference frame of an object. It is therefore aligned to the objects principal components. A bounding volume is a closed volume that completely contains the object[3]. After a transformation of the object from its frame and the world frame the dimension of the bounding box are obtained by computing the maximum and minimum coordinates of the point cloud. In this way it is possible to have an approximation of the length, width and height of an object.

Convex Hull A convex hull of a point cloud P is the smallest 3D convex set that contains P. In Figure 5.6 an example of the convex hull for a point cloud is shown. The vertices are



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

Figure 5.6: Convex hull example

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

Collision Detection To understand if an object impeds 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 a certain one. The collision detection is therefore a crucial step for the predicates computation. There exist different techniques to assert if two objects are

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

colliding, all of them need a representation of the object, which could be for example 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*.

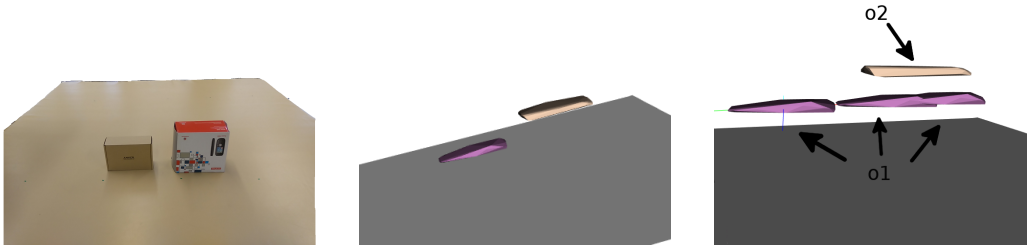
Given two objects A and B and their configurations $\mathbf{q_A}$ and $\mathbf{q_B}$, the discrete collision query returns a boolean value about whether two objects collide or not. Two objects collide if

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

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

To make the collision detection most collision libraries, before to use complex algorithm to detect collision between two shapes, they first check if the AABB of the objects intersect, if they don't the objects surely don't collide. If their AABB intersect the objects might collide.

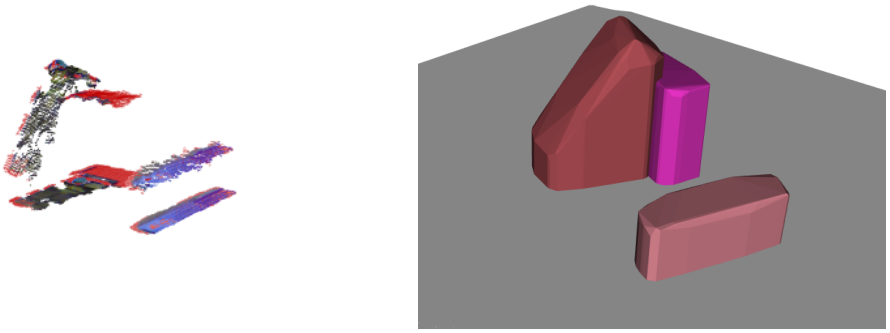
Objects Modelling The triangle mesh needed for the collision checking can be directly retrieved by the convex hull. But there is a consideration to take into account. The Kinect, given its pose, is only able to see mainly the top surface of the objects and not all the sides, and therefore we cannot apply directly the convex hull algorithm to the detected surfaces. If we would apply the convex hull on an object's surfaces, we will have likely the situation in which a surface S transformed in a configuration such that it will be above or below another one, and the collision detection will detect no collision since the two surfaces are not colliding. This is because we are totally missing the rest of information about the geometry of the object since we only know the objects surfaces.



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

Figure 5.7: 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 5.7b it is possible appreciating that we miss the information about the hidden part of the object. In Figure 5.7c a collision detection example is depicted. The convex hull is translated along a direction and no collision is detected with any possible translation since the two convex hull will not intersect.

From the Kinect's point cloud also the table plane is known, so all the information we have are: the table plane model and the segmented objects (mainly the top surfaces). If an human would be in the same pose of the Kinect, looking at the table, it will imagine that the objects are not floating surfaces, and he/she will deduce the objects shape from the shape of the top surface. Since in this work we assumed to work with parallelepipeds, the sides of the objects can be easily 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 more trivial 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.



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

Figure 5.8: Convex hull of the objects using their projections onto the table plane.

5.3. Actions Execution

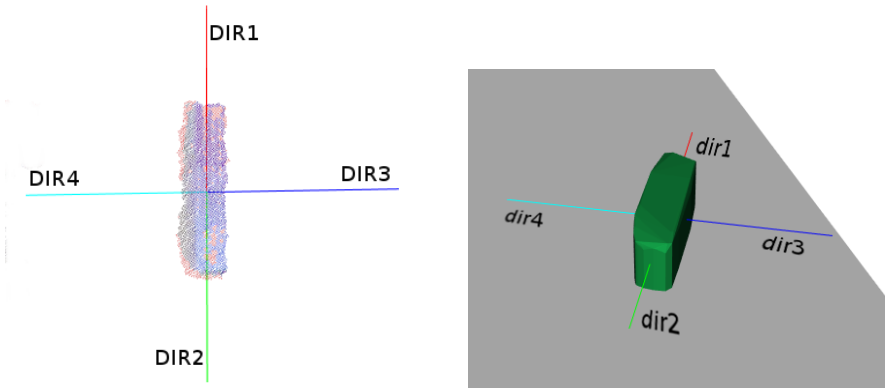
In this section the way the actions are executed is discussed, since the predicates computation depends on the way we decided to execute the actions.

5.3.1. Pushing

Pushing is a difficult action to execute when the goal is to move one object from one pose to another one. At the best of my knowledge there are still no methods in state of the art that faced this problem. 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 [29] [27] [28] , without taking care about the final position of the objects. Hermans er al. [25] 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. This means that the pushing action could move several objects and the pushing direction is not chosen accordingly the object to push but accordingly to the plane that separate the objects. This results in a pushing action without any guarantee that the pushed objects follows the desired path. Grasping , although is a difficult task, has already an advanced state of the art regarding finding grasping pose also for complex objects. Somehow grasping is easier because it does not consider the whole object geometry but it is focused on looking for local futures that can be interesting for grasping the object (i.e. antipodal points). When we push an object all the object geometry has to be taken into account in order to find a suitable pushing direction. Moreover, humans push objects with a certain initial direction chosen accordingly the object geometry and then adapt the pushing direction accordingly to the error with respect the desired one. This would imply the implementation of a controller which is out the scope of this thesis.

We are considering 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 5.5b), but he/she also could push it along the diagonal thanks to the several amount of freedom of the hands. Inspired by this consideration, we decided to consider as possible directions to push an object its first two principal components, the third is orthogonal to the object and it is not of interest for pushing. In particular, there are two senses for each direction, so in total we have 4 possible pushing directions per object, as depicted in Figure 5.9.

Another things to take into account is that the principal directions are not parallel to



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

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

Figure 5.9: Example of pushing directions.

the table plane. The Kinect could see some sides of the objects, and these sides will affect the direction of the principal components. An object which stands on top of a table will be obviously pushed along a direction parallel to the table. As previously described in Section 5.2 a point can be projected on to the table plane, but a point and a vector (the principal components are vectors) have the same representation, and therefore we can apply exactly the same equations to obtain the projections of the principal components onto the table plane. So the pushing directions considered are not the one obtained by the PCA but their projections.

Next, having some pushing directions along with the robot will push the objects, the pose of the end effector (i.e. the gripper) to push the object has to be decided. The pose has been chosen accordingly to the shape of the end effector. In Figure 5.10 is possible observing the profile of the gripper mounted to the last link of the robot, highlighted by the blue color. Such last link has a circular shape and the gripper's deep is less then the last link. It is undesirable pushing an object with a circular, or spherical, shape for the end effector because there is more probability to make the object no follow the desired path. 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. As final consideration, we don't want that the last link of the robot touch the objective object since we want a pushing action accurate as most as possible.

Figure 5.10: Gripper's profile view and robot's end effector.

Knowing also the height of the objects retrieved by its AABB, it is possible having a pose for the gripper is such a way that the end effector does not touch the object. The gripper's pose, relative the object, is computed in manner to locate the red point of Figure 5.10 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 which is tangent to an object without colliding with it.

Due to the limited opening width of the gripper (8 centimetres) the object the robot is going to manipulate have small width. This means that when pushing along the principal

Figure 5.11: Profile view of a desired pose for pushing an object.

axis, the object's width is likely small. Such a situation is depicted in Figure 5.12a. Pushing in such a way the gripper will likely push also the black juice box. Therefore when pushing along the principal axis the pose is chosen to be the one in Figure 5.12b. Of course is more stable a pushing pose like the one in Figure 5.12a, since the contacts point (the fingers) are more distant. For this reason that pose is used only when pushing the objects along a direction orthogonal to the principal axis.

(a) **(b)**

Figure 5.12: Possible pushing poses for push an object along its principal axis - In Figure 5.12a the closing direction of the gripper is orthogonal the pushing direction, and for the case depicted in the figure the gripper will likely push also the black juice box. In Figure 5.12b the closing direction of the gripper is parallel to the pushing direction.

Having a the projections of the principal components, the table normal and the gripper closing point coordinates it is possible defining a transform defined by the following rotation matrix and translation vector:

$$R_{dir1} = \begin{bmatrix} dir2_X & dir2_Y & dir2_Z \\ dir4_X & dir4_Y & dir4_Z \\ n_x & n_y & n_z \end{bmatrix}^\top \qquad T = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \tag{5.2}$$

where $dir1_X$ refers to the x coordinate of the vector that defines the direction 1, n is the table normal and c is the desired tool central point. Eq. 5.2 refers for a pushing along direction 1, where the gripper's closing direction has to be parallel to the pushing direction. The rotation matrix R_{dir3} for pushing along direction 3 would be the same of R_{dir1}.

As previously said, the planner consider to push the objects at infinity, in reality the robot has to push the objects for a finite length. Since the planner has no geometric information, the pushing length is chosen accordingly the dimension of the AABB relative

to the pushing direction. For instance, if the robot is going to push the object 1 along direction 1 the length l of the pushing action is $l = k \cdot AABB_1.length$, where k is a gain factor (1 in our experiments) called *pushing step*. This is a big limitation since the robot will push an object accordingly to the manipulated objects, not to surrounding ones.

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

When the robot approaches the pushing pose it could be that it collides with other objects. It would be suitable to use *MoveIt!* which can built an octomap representation of the scene and find a path avoiding collision with the scene. Although this technique is very powerful and correct for this application is quite time consuming. We avoid this by simply considering a pre pushing pose which has the same pose of the pushing pose but translated, accordingly the table normal, 10 centimetres from the pushing pose. After the execution of the pushing action the robot goes to its *home* pose (depicted in Figure 2.5a) in order not to fall inside the Kinect's view. When it goes to home it might happen that it collides with some objects, so also for the final pose with consider another one translated, accordingly the table normal, 10 centimetres from the last pose. In this way the pushing trajectory is defined by a total of $n + 4$ poses.

5.3.2. Grasping

There exist an advanced state of the art regarding grasping. Despite this all the techniques of grasping are very computationally expensive. Many of them rely on the identification of the shape of the objects and then a set of pre-built grasping poses is returned and they can be very robust as the method proposed by Brook et al. [12]. Other techniques rely on the identification of local features which can state if a grasping pose is feasible or not. Two very goods grasping planning algorithms, which deal with novel objects, published past year are AGILE [43] and HAF [22], despite this how shown in [15] they are not so robust and they are computationally expensive and not suitable for this thesis. In order to have a faster planning algorithm we considered a very simple approach to grasp the objects, which is suitable only with the kind objects we are going to interact with. Despite this the planner present by this thesis can be directly integrated (with just few modifications) with several grasping algorithms.

The idea is to grasp the object in manner that the gripper's closing direction is orthogonal the principal axis of the object. The approaching direction of the gripper is given by the third principal component of the object. Then the gripper also is centred accordingly the centroid of the object. In this manner a single grasping pose is obtained for each object.

Add exactly how the gripper position is computed

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

The grasping action, in its total is composed by the following set of actions:

1. Reach the pre grasping pose
2. Open gripper
3. Reach grasping pose

4. Close gripper

5. Reach pre grasping pose again: this is done in order to avoid collisions between the grasped object and the other ones

6. Go to the dropping pose: the object will be dropped into a bin

5.4. Predicates Computations

In Chapter 4 the predicates used are described, in this section their computation is presented in detail.

Once the objects have been segmented, we have one point cloud per object, next we have to retrieve the symbolic predicates from the objects configuration.

5.4.1. Predicate: `block_grasp`

The `block_grasp o1 o2` predicate refers the fact that object `o1` impedes `o2` 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 `o2`, and check if it collides with the other objects. In figure 5.13 such procedure is shown and in Algorithm 2 the pseudo algorithm is described in detail.

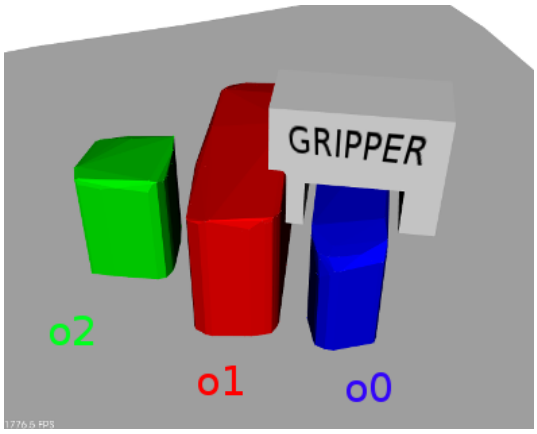


Figure 5.13: Visualization of the computation of `block_grasp` predicate for object `o0`. The opened gripped model is transformed to the grasping pose for object `o0` and it is tested if the gripper mesh model collides with the other objects, in this it collides with `o1`.

Notice that this method implies to check for collision between the gripper and objects that might be very far from the interested object, i.e. there is no need to compute the collision detection. Despite this, how explained in Section 5.2 the most collision detection algorithm first check if the AABB of the objects intersect. This is a computationally cheap operation, and only if their AABB 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 8 milliseconds. This for two main reason, the objects are far between each other and their AABB do not intersect, or in case they intersect the gripper mesh model is relatively simple compared to the convex hull of an object, this makes the collision detection algorithm faster because it has to check for the intersection of less triangles.

Algorithm 2 Computation of `block_grasp` predicate. O is the set of objects (convex hull retrieved with the projection onto the table plane) and G_{poses} is the set of grasping poses of all the objects.

function ComputeBlockGraspPredicates(O,G_{poses}) `block_grasp_predicates` = NULL
  for all $A = objects \in O$ **do**
    $openGripperMesh \leftarrow$ trasnformGripperModel($G_{poses}(A)$)
    for all $B = objects \in O$ **do**
      if $A \neq B$ **then**
        $collision \leftarrow$ isThereCollision($openGripperMesh$,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

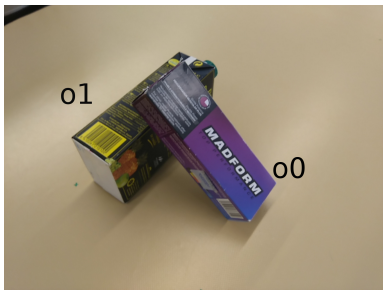
5.4.2. Predicate: `on`

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 on 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 [20] [36] [21] the authors proposed some approaches to enhance 3D recognition and segmentation results to create and maintain a consistent world model involving attributes of the objects and spatial relations among them. Their researched focused on modelling the world for manipulation planning tasks. They do not consider scene like the one of this thesis but simpler ones such as a pile of cubes above each other. What can be directly used from their work is the computation of the `on` predicate. The *on* relation for a pair of objects is determined by checking whether their projections on the table plane overlap. This predicate was not a relevant part of their work and they did not provide to much information about its computation. Therefore our implementation for the `on` predicate is based on their idea with some modifications.

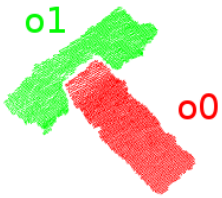
Our idea is based on the fact that an object which stands on top of another one occludes some parts of the object below. While the one below does no occlude any part of the top object. Let's consider the scene in Figure 5.14a, the object `o0` occludes some parts of object `o1`. The projections P_0 and P_1 onto the table plane of `o0` and `o1` are respectively the red and green ones in Figure 5.14b. This means that the convex hull C_{P_1} of the projection `o1` intersect 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 5.14c and 5.14d).

Although this method works fine to compute the `on` predicate it has the limitation that its scope is only for objects for a rectangular shaper. An object that has a form of L has a convex hull that includes big free areas where an object could lie.

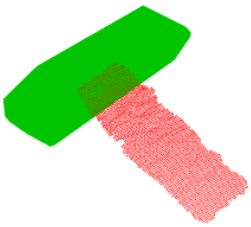
It is important to take into account also that actually the edges of the occluded parts of the below object, once projected, could be at the same position, of some projected edges of the top object. This could be dangerous for the computation of this predicate. Therefore a threshold is added. Focusing the attention on Figure 5.14c it can be appreciated that



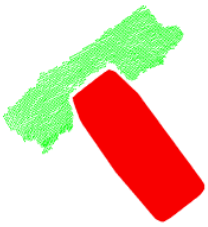
(a) Scene



(b) P_0 & P_1



(c) C_{P_1} & P_0



(d) C_{P_0} & P_1

Figure 5.14: Visualization of the computation of the `on` predicate. Figure 5.14b show the real image, Figure 5.14b shows the projections of the objects onto the table plan while Figures 5.14c and 5.14d represent the two step strategy to compute the `on` predicate.

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

$$\texttt{(on o0 o1)} = \begin{cases} True, & length(C_{P_0} \cap P_1) > th_0 \quad \wedge \quad length(C_{P_1} \cap P_0) < th_1 \\ False, & otherwise \end{cases} \tag{5.3}$$

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

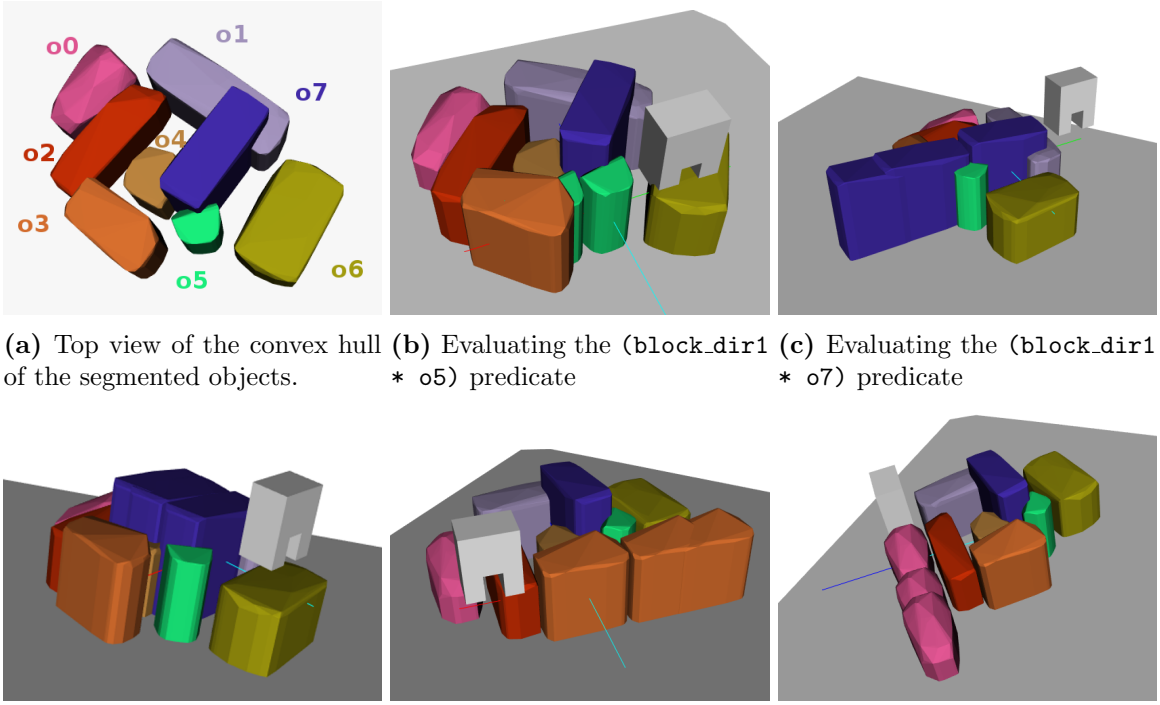
The formula 5.3 is then evaluated for every possible combinations of objects, that is $n(n-1)$ times, where n is the number of objects. Despite this, its computation is very fast. For the example in Figure 5.14 its was evaluated just 2 times since there are only 2 objects and it took 3 milliseconds, that is $\approx 1.5 \frac{ms}{pair\ of\ object}$. For instance, for a complex scene with 10 objects the total time devoted to compute this predicate would be $10 \cdot 9 \cdot 1.5 \approx 135ms$ (the $\approx$ symbol is due to the fact that the computation time depends on the number of points of the objects and the complexity of their convex hulls).

5.4.3. Predicate: `block_dir`$_i$

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

Object `o1` can impedes 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 ocnvex hull C_0 is translated along the considered direction

until reach 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. To do that we could use a continuous collision detection algorithm or a discrete one. The continuous method suppose that given an object, its initial, its final pose and the other objects in the environment it will detect a collision if a collision occurs from the initial to the final pose. This algorithm has the advantage to be accurate but we should repeat it separately for each object sine we are not interested to see if there is a collision, but we are interested in knowing what is the object it is going to collide with. We decided therefore to use the discrete strategy, that is we consider several poses between the initial one and the final one, including the final one, and for each check if it 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 5.15: Visualization of the computation of `block_dir`$_i$ predicates. Evualuating the `(block_dir1 * o0)` means that the algorithm is evaluating for all the objects, except `o0`, if they collide with `o0` when pushed along direction 1.

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 AABB 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 5.3.1.

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 impedes 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 relative the desired pushing direction of the object to push and check for collision with the other objects, as similarly done for the computation of the `block_grasp` predicate.

In Figure 5.15 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 how done for the object. We decided to neglect this and check only for the initial pose in order to relax the planner and not to make it too much conservative. This means that during the pushing action the robot might actually move more than one object. Despite this relaxing strategy the algorithm showed to work perfectly and case in which the gripper moved more than one object were really rare, at least we the objects we used in our experiments.

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

```
function ComputeBlockDirPredicates(O,P_d,P_poses,P_path)
block_dir_predicates ← NULL
  for all A = object ∈ O do
    for all d = directions ∈ P_d(A) do
      for all p = poses ∈ P_path(A, d) do
        for all B = object ∈ O do
          A_T ← transformObject(A, p)
          if A ≠ B then
            collision ← isThereCollision(A_T,B)
            if collision then
              block_dir_predicates ← AddPredicate(block_dir B A)
            end if
          end if
        end for
      end for
      closedGripperMesh ← trasnformClosedGripperModel(P_poses(A, d))
      for all B = object ∈ O do
        if A ≠ B then
          collision ← isThereCollision(closedGripperMesh,B)
          if collision then
            block_dir_predicates ← AddPredicate(block_dir B A)
          end if
        end if
      end for
    end for
  end for
return block_dir_predicates
end function
```

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 $\mathcal{O}(n \cdot n_{directions} \cdot n_{poses} \cdot (n-1) + (n-1) \cdot n_{directions})$.

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 5.15 (considering to push the objects 1.5 times the AABB dimension relative to the pushing direction) was $\approx 1.115 seconds$, were $\approx 0.869s$ were dedicated to check if the objects will collide when move while $\approx 0.239s$ were dedicated

if the gripper collides with some objects.

Photos to take: - Robot pushing (vertical and parallel mode)

6. Implementation

Talk about ROS and real robot.

For the collision checking the Flexible Collision Library (FCL) [37] has been used. This library allows to define the collision problem in a simpler manner than other more famous collision libraries such as *Bullet* [8], and it can work with different objects shapes such as box, spheres, cone, convex, mesh and octree. The main library used in this work is the Point Cloud Library (PCL) [6], which allows some methods to create an object shape from a point cloud.

1. Nodes
2. Ros graph
3. simulation
4. PCL - FCL

7. Experiments

In this chapter the performance of the proposed task planner for the considered task is evaluated by performing several experiments with the aim to study the quality, its benefits and its limitations.

To get the quality of the task planner the strategy defined in the article [10] by Amignoni et al. In that article they proposed a general method to evaluated the quality of a task planner. First they define a functionality benchmark[1] thought 4 elements:

- Description: a high level, general, description of the functionality
- Input/Output: the information available to the module implementing the functionality when executed, and the expected outcome.
- Benchmarking data: the data needed to perform the evaluation of the performance of the functional module.
- Metrics: algorithms to process benchmarking data in an objective way.

Regarding the benchmarking of a task planner those elements are defined as:

Description: Given a initial state and a goal state find a set of actions to reach the goal.

Input/Output: The inputs are the PDDL domain and problem descriptions. The output is a plan expressed as sequence of actions.

Benchmarking data: Some data has to be supplied by the robot, including: the input given to the planner, the output (i.e., the plan) delivered by the planner, and the performance data for the planner (e.g., memory and time needed). According to the planning approach, other data might be required. Other data will be possibly collected when a plan is actually executed, including: time needed to execute each action and time needed to execute the whole plan.

Metrics: Metrics are mainly related to good plan quality and to solving time. The target of benchmarking task planning is twofold: to asses the quality of the plan produced (e.g., in terms of the cost for executing it and its likelihood to succeed) and to assess the quality of the planning process (i.e., how fast is the plan generated and how much resources are needed to do so). Possible metrics are:

  - *Levenshtein distance*[2] is a metric that specifies the distance between two strings, in this case a plan can be though as a sequence of character, where each action

[1] A functionality benchmark is a benchmark of specific robotic task. In the case of this thesis a benchmark of the task planner.

[2] https://en.wikipedia.org/wiki/Levenshtein_distance

with a determined object of interest is a character of the string. The Levenshtein distance between the two lists is calculated to measure the quality of the plan. Levenshtein distance $L_{A,B}(|A|, |B|)$ between two lists A and B is calculated as

$$
\begin{aligned}
L_{A,B}(i,j) &= \max(i,j) \qquad if \quad \min(i,j) = 0 \\
L_{A,B}(i,j) &= \min(L_{A,B}(i-1,j)+1, \\
&\qquad\qquad L_{A,B}(i,j-1)+1, \\
&\qquad\qquad L_{A,B}(i-1,j-1)+1_{a_i \neq b_i}) \qquad otherwise
\end{aligned}
$$

where $|A|$ is the length of plan A, and $1_{a_i \neq b_i}$ is equal to 0 if the action a_i is the same of action b_i. This metric will be used to evaluate the distance between the optimal plan returned by the planner in the first frame and the real executed plan.

- Number of actions correctly performed in the correct order (e.g., number of objects delivered to their correct destination).
- Time required to construct a plan and time to perform it.

We will also evaluate the difference of executing the initial plan and executing the plan with the replanning technique in terms of time and number of correct actions. As well also the number of bad, or dangerous, object manipulations due to the lack of replanning, lack of probability and geometric constraints in the planner .

It is important highlighting that what has been done is not a benchmarking since no comparison is done with other state of the art planners, but the commented guideline for benchmarks has been used in order to get some useful metrics in order to assert the quality of the planner.

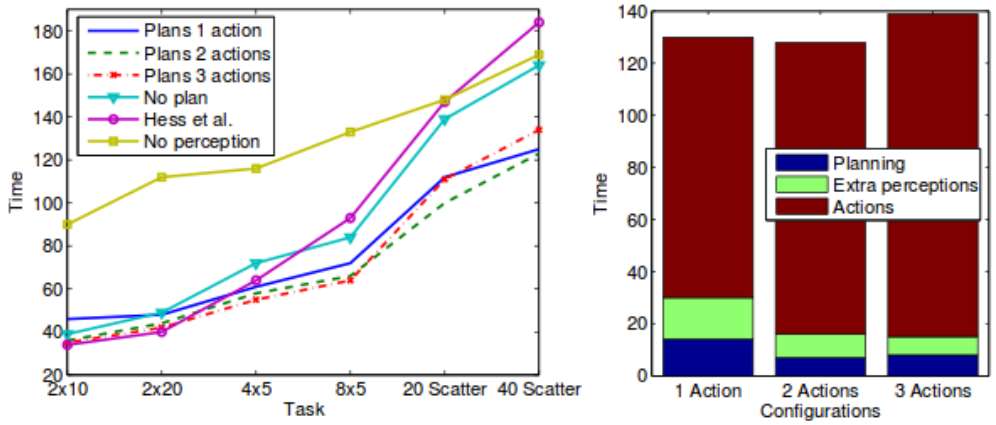
Kinds of possible experiment to do:

- show experiments first trying only separating the object (infinite loop behaviour) and then coupling the grasping action.
- Show the results of the planner with the pushing length chosen not accordingly to the surrounding objects but accordingly to the pushed one. From that comment the improvements.
- Show normal experiment where everything works
- Show situation why we are going to consider the risk as cost in the action.

Metrics to considers:

- number of objects
- segmentation time
- Total solution time
- planning time
- total solution
- Inverse kinematics solution time (this will probably used to justify the backtracking method)
- Predicates time computation (maybe add collision checking time in)

Find a way to get the computational complexity.



(a) Cleaning lentils

(b) Cleaning 40 lentils

Figure 8: (a) Skill of moving lentils to a container. The results shown are the mean over 5 runs. (b) Time for the case of moving 40 scattered lentils to a container. Extra perceptions include the movements and perceptions needed to tackle occlusions by taking the arm out of the field of view (see Section 4.2.3).

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

Bibliography

[1] Description of the shop project https://www.cs.umd.edu/projects/shop/description.html .

[2] Gourmand Home Page.

[3] PROST Home Page.

[4] SHOP Home Page.

[5] Fast Downward Home Page.

[6] http://www.openperception.org/ - open perception.

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

[8] Bullet `http://wiki.ros.org/bullet`, 2015.

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

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

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

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

[13] Stéphane Cambon, Fabien Gravot, and Rachid Alami. A robot task planner that merges symbolic and geometric reasoning. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 895–899. IOS Press, 2004.

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

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

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

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

[18] Mehmet Dogar and Siddhartha Srinivasa. Push-grasping with dexterous hands: Mechanics and a method. In *Proceedings of 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*, October 2010.

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

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

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

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

[23] Keliang He, Morteza Lahijanian, Lydia E. Kavraki, and Moshe Y. Vardi. Towards manipulation planning with temporal logic specifications. In *2015 IEEE Intl. Conf. Robotics and Automation (ICRA)*, pages 346–352, Seattle, WA, May 2015. IEEE.

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

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

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

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

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

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

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

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

[32] Tomás Lozano-Pérez and Leslie Pack Kaelbling. A constraint-based method for solving sequential manipulation planning problems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 3684–3691. IEEE, 2014.

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

[34] Tekin A Mericli, Manuela Veloso, and H Levent Akin. Achievable push-manipulation for complex passive mobile objects using past experience. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 71–78. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

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

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

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

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

[39] R. Reiter. A logic for default reasoning. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 68–93. Kaufmann, Los Altos, CA, 1987.

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

[41] T. Siméon, J.-P. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. *Advanced Robotics*, 14(6):477–493, 2000.

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

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

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