SAPIENZA
UNIVERSITÀ DI ROMA

Dealing with anomalies in robot-enabled industrial quality control laboratory

Master's Thesis in Artificial Intelligence and Robotics.
Department of Computer, Control, and Management Engineering:
"Antonio Ruberti"

Candidate

Mario Fiorino
ID number 1871233

Thesis Advisor

Prof. Luca Iocchi
Sapienza University of Rome

Co-Advisor

Dr. Fabio Zonfrilli
Research & Development
Procter & Gamble - Belgium

Academic Year 2020/2021

Thesis defended on 17 January 2022
in front of a Board of Examiners composed by:

Prof. Luca Iocchi (chairman)
Prof. Silvia Bonomi
Prof. Irene Amerini
Prof. Thomas Alessandro Ciarfuglia
Prof. Christian Napoli
Prof. Leonardo Querzoni

Dealing with anomalies in robot-enabled industrial quality control laboratory
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: mario.fiorino@gmail.com

"Always take a job that you suppose is too big for you"

Contents

Chapter 1

Introduction

This thesis work was born and developed within the wider European project: AIPlan4EU; from the collaboration of the company Procter & Gamble Services Company - Belgium and Sapienza University of Rome.

AIPlan4EU project is funded by the European Commission, was launched in January 2021 and will have a duration of 3 years; his consortium comprises 16 European partners from different industries and with various expertise (research groups and companies). AIPlan4EU aims to bring the most advanced planning technology Europe has to offer to companies, through the AI4EU Platform[1], demonstrating how this technology can be adopted in different scenarios, and encouraging the research in AI planning in a modern, application-oriented fashion. To do so, use-cases from diverse application areas drive the design, and include several available planning systems that can be selected to solve practical problems [2].

For this thesis, AI Planning use-cases was provide by Procter & Gamble, exactly by Fabio Zonfrilli R&D Principal Scientist Robotics and Automation; within the context of industrial quality control. This consists of the executive procedure for the analysis of the laundry pouch (soluble capsules contain liquid and powder) performed by robotic arm Universal Robot 5 with a 2F gripper, in a dedicated laboratory, using different measuring instruments. Furthermore, it was asked to lay the foundations for an AI planning framework that included within it the human component: whose ultimate purpose will be to create a standardized system mixed human-robot for product testing procedures.

I had a role as a forerunner in this context: the first phase of my thesis work consisted in building the simulator, that allowed to experiment with the AI plans that I thought; this simulator will also be the starting point for future ones. Working at first on the simulator, to then pass on the real robot in real scenario, allows to reduce the high costs of design, experimentation and debug code, which would entail working directly on the hardware in real world situation.

Starting from scratch, for guaranteeing portability and self-sufficiency in most computing environments, i created a Docker image (a state-of -the-art containerization technology) contains custom ROS (Robot Operating System: framework robot application) packages for the UR5 Robot with a gripper Robotiq 2F-85, and Gazebo (3D robotics simulator for ROS system) within which I reconstructed the laboratory working environment in its fundamental aspects.

The second phase of my thesis work consisted in the conception, implementation and verification test of AI plans, formulated through Petri Net Plans and Plan

[1] https://www.ai4europe.eu/

[2] For more information: https://www.aiplan4eu-project.eu/

Execution Interface frameworks. Regarding formulation and implementation of a complex plan, these technologies guarantee modularity, high expressiveness (accessible also for an users are not experts of the robotic system) and flexibility in case of updates and structural changes. One of the most interesting parts of this phase was to work on recovery procedures at the nominal plan: robotic systems are not yet "perfect" machines, in the real world failures or unforeseen situations can happen. To deal with this type of problem, the strategy used was that of: at first, define a recovery plan thanks to which the robot autonomously try to resumes the nominal plan from where it left off; then if this should not be sufficient, require for human intervention (Human-in-the-loop concept). The MODIM framework was used to optimize human-robot interaction.

This document, in addition to presenting the thesis work, also aim to be a basis user guide for future projects and experiments.

Keywords: AIPlan4EU, AI Planning, Universal Robot 5, Docker, ROS, ROS Action, Gazebo, Petri Net Plans, Plan Execution Interface, MODIM, Recovery Procedure, Human–robot interaction, Human-in-the-loop.

1.1 Chapters presentation

In Chapter 2 an introduction to AI planning, the basic ideas of computational learning technique in the field of robotic behaviors, and Learning from Demonstration paradigm are briefly presented. With this in mind, the description of the main papers which have inspired the formation of this thesis work and its future developments are outlined.

In Chapter 3 the software technology used for the building of the robot simulator and its virtual lab environment: Docker, ROS, Gazebo, is described. Their advantages and disadvantages, how they have been used to develop this thesis, and the results obtained are shown. Then are presented an essential description of the main real laboratory tools: Universal Robot 5 and Robotiq 2F-85 gripper, digitized in the simulator. The chapter ends with Petri Net Plans and PlanExecution Interface frameworks, with which the plans were implemented.

In Chapter 4 the nominal plan as a whole; the result of analysis the cases of the fail: their categorization into four areas; and finally the principles, centered around the concept of Human-in-the-loop, underlying the proposed recovery solution are illustrated.

In Chapter 5 the implementation phase of the plans and its results are presented. Starting from the design of simple plans, more complex plans have been formulated, and, at the end, proposed directions and objectives for future works.

In Chapter A a series of technical aspects, used in this thesis, are presented; for the reader who would like to know more.

Chapter 2

Theoretical Starting Point

2.1 AI planning

AI planning is a field of Artificial Intelligence that concerns the the computational formulation of a plan: it is an organized sequence of actions performed by an automatic planner (intelligent agent, e.g. it can be autonomous robots or unmanned vehicles) which, interacting with the environment(world), starts from an initial state of the world, and reach a predetermined objective: a state of the world in which specific conditions are met. Each action requires a certain number of preconditions to be verified in order to be applied to a particular world state. Upon application, each action produces effects: changing the world state. The motivations for investigating these studies are well explained by Ghallab, Nau, Traverso [17]:

One motivation for automated planning is very practical: designing information processing tools that give access to affordable and efficient planning resources. Another motivation for automated planning is more theoretical. Planning is an important component of rational behavior. If one purpose of AI is to grasp the computational aspects of intelligence, then certainly planning, as the reasoning side of acting, is a key element for such a purpose. The challenge is to study planning not as an independent abstract process but as a fully integrated component of deliberative behavior. An important combination of the practical and theoretical motivations for automated planning is the study and design of autonomous intelligent machines

A formalization of the fundamental concepts of AI planning can be shortly described in the following way. A planning problem consists of:

- The initial state of the world
- The goal state to achieve
- The state transition system responsible for changing the world state by action application

The state transition system in the previous definition in turn is composed of

- Set of states[1].
- Set of actions. Each action is defined in the form of the unit of three lists:

  pre list - predicates, if verified, allow the applicability of the action;

  add list - predicates that become true (in this sense are "added") in the current state by the action application;

  del list - predicates become false (deleted) in current state as a result of the applied action.[2]

  The listing of all the applicable actions in a domain is called the action model.
- the state transition function : from (State,Action) to (State)

[1]Note concerning the representation of a state: it is necessary to provide the planner with a model of world on which it operates; the states of world are usually represented in sentences form, e.g.: on(x,y) AND Clear(a) AND hold(r,k) AND ...

[2]Note : Sometimes in planning language like STRIPS (Stanford Research Institute Problem Solver), *add* list and *del* list are represented as an unique *effect* list with positive and negative formula. Note further, STRIPS is based on assumption : everything that is not specified in *add* list and *del* list or simply *effect* list remains unchanged in state

All this elements contribute to the formulation of a plan: given in input an initial state of the world, goal, and an action model; a plan is a sequence of actions that drives the system from the initial state to the goal.

The main drawbacks encountered in the development of a plan is the creation of action models for complex environments(alias domain): while it is possible to codify action models for simple game-domains, it takes a lot of time, effort, skill, experience, and sometimes it is impractical to do so for some complex real-world domains. Real world planning domain models are difficult to develop, debug and maintain; and there is also no "one size fits all" strategy. For these reasons it is particularly appropriate to develop systems for automatically acquire (or even optimize) the planning models using Machine Learning techniques, which allow learning of the underlying action model from traces produced as a result of plan execution.[3]

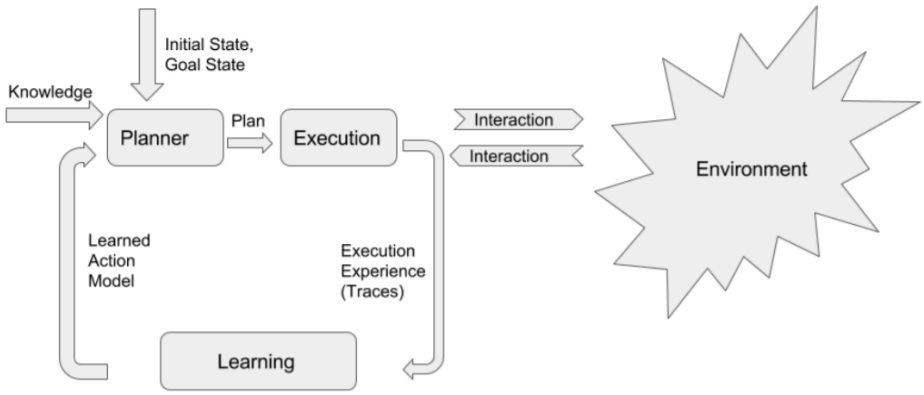

Figure 2.1. Learning planning action model diagram [16]

[3]Note: the model obtained can ideally be re-injected into the planner for further planning purposes.

2.2 Computational learning approach

Currently, a highly successful machine learning technique to design sophisticated and hard-to-engineer behavior for a robot interacting with the environment, is the Reinforcement Learning; a wide variety of problems in the field of robotic behavior may be formulated as ones of Reinforcement Learning.

Reinforcement Learning (RL) is a computational learning approach that deals with learning in sequential decision making problems conditioned by feedback. It is one of three main machine learning paradigms, with supervised learning (that need labelled input/output pairs to infer a mapping between them) and unsupervised learning (that analyzes a large amount of unlabelled data to captures patterns from them). The intuitive idea behind Reinforcement Learning methodology is exhaustively expressed by Sutton and Barto in introduction of their book [20]:

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves.

RL algorithms are characterized by four basic element: Agent: a program to train for a certain purpose. Environment: the "world" in which the agent can stay and performs actions, practically a collection of state signal. Action: a "movement" made by the agent, that allows to switch from one state to another in the environment. Reward signal (o simply Reward): the evaluation of an action, numerical feedback from environment. Key point of this algorithms is: what to do—how to map state to actions so as to maximize a numerical reward signal over the time. The agent must discover which actions yield the most reward by trying them: the data is generated with a trial-and-error search during training within the environment, and marked with a label. The mapping from states to actions(to be taken when in those states) is called policy. The policy can be described as one function, or lookup table, or through more complex computational process. Saying in other words, solving a RL task means, finding a policy, called optimal policy, that maximize the total reward over the long run.

This type of learning allows to automate the resolution of complex control and interaction problems with the environment (in some cases not solvable with explicitly programming), potentially (from the theoretical point of view), proposing the best possible solution.

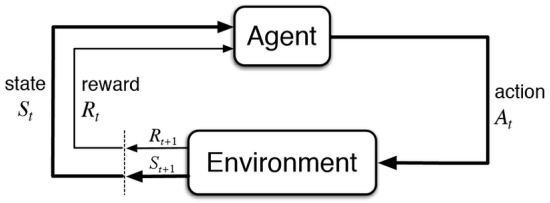

Figure 2.2. The reinforcement learning framework [20]

In most of the control problems in the field of robotics, the environment (the

real word) in which the robot interacts, can be thought of with an acceptable simplification called Markovian environment: this means that a future ("next") step of a proceeding is conditioned only by the current state, the present captures all relevant information from the history (practically: once the present state is known, the information about the past states may be thrown away!). This condition is also know as "Markov assumption".

Markov Decision Process (MDP) is mathematical framework that typically models Reinforcement Learning environment. In time discrete and stochastic control processes, MDP is a 5-tuple $\langle S, A, T, R, \gamma \rangle$:

- S is a finite set of states s
- A is a finite set of actions a
- T is a state transition matrix: $T_a(s, s') = Prob(s_{t+1} = s' | s_t = s, a_t = a)$, tell us the probability of ending in the state s' after executing action a in state s
- R is a reward function: $S \times A \to \Re$, the immediate reward received (a real number) with execution a action in a certain state s
- $\gamma \in [0, 1]$ is the discount factor, is a constant that takes into account the possibility of non-immediate rewards: in the case $\gamma \approx 0$ the agent will be "greedy" by considering predominantly the immediate rewards, opposite the case $\gamma \approx 1$ where the agent will pay attention to the long-term reward ("far-sighted" evaluation))

Given a MDP tuple, a policy of an agent can be define as is a distribution over actions given states, formally:

$$\pi(a|s) = Prob(a_t = a | s_t = s)$$

The optimal policy is defined as policy that chooses actions over time so as to maximize the expected value of the function return G_t :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} ... = \sum_{k=0}^{\infty} \gamma^k \ R_{t+k+1}$$

called the total discounted reward from time-step t, over a potentially infinite event horizon. To find the optimal policy is often used the concept of state-value function $V_\pi(s)$:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s]$$

it is the expected return G_t starting from state s, and following policy π. Practically, the state-value function $V_\pi(s)$ gives an evaluation of state s when is following the policy π. For compute the state-value function for a given policy, it is possible to represented it by the Bellman Equation:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \left(\mathbb{E}[R_{t+1} | s_t = s, a_t = a] + \gamma \sum_{s' \in S} T_a(s, s') V_\pi(s') \right)$$

Combining into one system all the Bellman equations in an MDP with n available states, we obtain n linear equations for the n unknown value functions. Solving this linear equations system it is possible to get the value functions; obviously direct solution are only feasible for small MDP.

In real word robotic control, can be difficult (in certain scenarios infeasible) to specified a reward function that allows to learn complex tasks (more properly: design a behavior by a reward function that encodes the behavior's objective). In addition to this, the large number of possible actions, combined with an even greater number of possible states, increases a lot the difficulty of learning an optimal policy [36].

2.3 Learning from Demonstration

Robot Learning from Demonstration (LfD) (also know as Robot Programming by Demonstration) is a learning paradigm that allow a robot system to perform new tasks from observations of a human's own performance[4]; all this without using any traditional programming scenario (means: reasoning in advance and coding a robot controller in way that it is capable to deal with any situation it may encounter during the task).

The key idea of this approach is deriving a policy, from examples provided, that reproduce the expert's behavior (the hypothesis that the demonstrations given in input are successful demonstrations of the desired task, is one of the basic assumptions of this paradigm). More specifically, LfD can be divided in two phase:

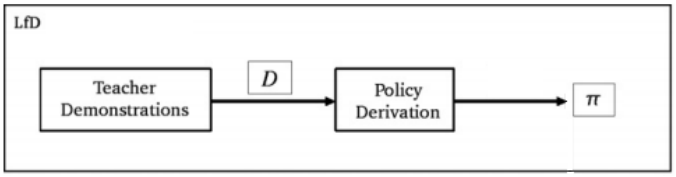

Figure 2.3. Learning from Demonstration framework.[14]. D symbol stands for data set, π for policy.

1. Build a data set: gathering a sequence of state-action pairs (examples) obtained by recording teacher executions of the desired task. In this phase how demonstrations are acquired play a central role; most generally, demonstration approaches can be grouped into two big categories :

   I) Kinesthetic and Teleoperation Teaching; in first the robot is physically moved and guided through the desired task by the humans; in second the demonstration is performed via an external input to the robot, like: a joystick, graphical user interface, or other remote control devices (may also be performed through speech dialog) allowing to train the robots from a distance. In both case, teacher actively operates on the robot, and the states/actions experienced during the interaction are recorded directly through its on-board sensor in data set[5].

   II) Passive Observations; in this case human performs the demonstration using their own body[6] (the robot takes no part in the execution of the task);

[4]Note: Currently this technique mainly refers to the human being, but this does not exclude that the teacher of the task could be an animals or other robots or a computer simulation

[5]States/actions recorded within the data set are exactly those that the robot would observe and execute.

[6]Sometimes wearing motion sensors to facilitate tracking

through sensors (typically vision-based), external to the executing platform, are recorded the human executions. Evidently, in this case the states/actions recorded are different those that the learner would observe and execute. For this reason, it is necessary either encode or learn a mapping from the human's states/actions to those executable by the robot (note that this entails a new source of uncertainty for the learner).

(a) Kinesthetic (b) Teleoperation (c) Observation

Figure 2.4. Showing examples of robot demonstrations from categories described above [3]

2. Deriving a policy: processing the examples from the previously data set for generating a policy. Basically, can be identify two main techniques to deriving a policy from the demonstration data set:

   I) Mapping Function Approach; the goal of this type of approach is to derive a function: state to action mapping, from statistical analysis of the data across demonstrations, so as to reproduce a behavior that are similar to those demonstrated by the human expert (this means to find a function that generalizes over the data set of available examples such that obtain valid solutions for similar states, not encountered during demonstration). Depending on whether the state input and action output are discrete or continuous, are used classification algorithm for discrete case (i.e. categorize input into discrete classes: where the input to the classifier is robot states and the discrete output classes are robot actions), and regression algorithm for continuous case (i.e. map demonstration states to continuous action spaces; the input to the regressor are robot states, and the continuous output are robot actions)[7]

   II) System Model Approach; i.e. learning a model of the world and deriving a policy from this information. This approach is typically formulated within the framework of Reinforcement Learning: from demonstration data determines the transition function $T_a(s, s')$, and from this derives a policy π . To derive a policy from this transition model, a reward function R, can be manually defines by engineers or learned from demonstrations[8]. The approach of an engineered reward function generally tends to set the reward value to zero, except for a few states, such as near the goal or around obstacles. This help the robot locating the rewards and prevent extensive periods of blind exploration. So the demonstrations is used to highlight the most rewarding areas of the state space. Regarding the learned reward functions, Inverse Reinforcement Learning (IRL) algorithms are used. This technique aim to extract reward function from observed exert trajectories, assuming that the expert is trying to optimize an unknown reward; i.e. it is assumed the demonstrations given do

[7]Note: typically classification algorithm can be applied at any level, whereas, regression approaches is applied to low-level motions and not high-level behaviors.

[8]Note : state and action selection are based on reward, practically it guides task execution; defining a reward function to accurately capture the desired behavior can present several difficulties

not contain errors and the expert aim to an optimal behavior. Depending on the complexity of the problem, the reward function to learn can be linear or not, in the latter case neural networks are used. Also note, since there could potentially be infinite reward functions that optimally describe the expert demonstrations, in order to get at a unique reward function, IRL methods consider optimization constrain such as maximum-margin (ie. the reward function is identified by maximizing the difference between the best policy and all other policies).

The learner performance are heavily conditioned by the quality and quantity of information in the demonstration data set. Argall et al. [14] identifies two main causes that lead to rather poor learner performance:

1. Scarcity of the data set, or the existence of areas of state space that have not been showed. This lead to the question of how the robot should work when it encounters a state without a demonstration. Two solution approaches are proposed: Generalize from existing demonstrations; i.e. improving and enhancing techniques of policy derivation described above. Acquire of new demonstrations, this means for the robot requesting the intervention of the teacher for additional demonstrations.

2. Poor quality data; i.e. teacher demonstrations may turn out ambiguous, unsuccessful or sub-optimal in certain areas of the state space. One of the most effective solutions is based on a feedback approach: if the learner is provided with feedback (e.g. provided via teacher) that evaluates its performance, this may be used to update its policy.

It is important to point out: LfD is not simply a record and replay technique. Interpret and generalizing human demonstrations is core to this approach. Solving this core research problem will likely revolutionize the next generation of robots.

The main motivations[9], which have contributed and continue to contribute to the development and diffusion of this technique, can be identified in:

- Explicit robot programming is complex, long, expensive, and sometimes tedious job: it may require the decomposition of the task into 100s of smaller different steps, and testing each step. If errors (or new circumstances) occur after the robot is deployed, the entire process may need to be repeated. In addition to that, though theoretically well-founded, the explicit robot programming depend heavily upon the accuracy of the world model: approximations such as linearization are often introduced for computational implementation, this, in more cases, inevitably leading to performance degradation. LfD methodology aim to eliminated, or minimized, this “hard” job.

- No expert knowledge is required: it is not necessary robot programming skill, but rather only the ability to demonstrate the task in the chosen manner; which is generally intuitive enough for everyone. This makes the robot system and the interaction with it more “natural”: particularly easy and accessible to everybody; this aspect can increase the chances of robots becoming more common and well accepted by a general public. And then, when a failure occur, the user needs only to supply more demonstrations, rather than calling for highly-skilled technicians help.

[9] Clearly combined with the technological development of high performing hardware and interface systems

- Considering the RL approach, the LfD methods reducing considerably the complexity of search spaces for learning an optimal policy. In many cases, trial-and-error learning strategy in a context with a huge number of possible state/action (like real-world) is impractical engineering with current technologies; and if feasible, the algorithms developed require a lot of time and memory for converge to an optimal solution, also for simple tasks. Focusing the search from a data set provided of "good" executions of the task, or conversely, by eliminating from the search space what is known as a "bad" executions; will be reduced the exploration needed, enhancing and accelerating the learning process.

2.4 Related work

Oldershaw et al. [1] propose a theoretical framework for learning plan recovery procedures (for improve the robustness of original plan) through human-robot interaction; assuming that the users are not experts of the robotic system, they don't know the robot's knowledge based, or programming.
The problem is formulated in these terms: given a robot linear plan (i.e. $\pi = \langle a_1, ..., a_n \rangle$), at certain point, during the execution of an action $a_i \in \pi$ occur a failure condition ϕ_i (i.e. the robot cannot determine a next state in the model and generate a new plan sequence). Human user is able to provide a demonstrations of how to recover from this failure status. This demonstration is formally a pair (p, a_s), where p is a recovery procedure (i.e. sequence of action $p = \langle \alpha_1, ..., \alpha_r \rangle$) that guide the robot to a state on which it can recover the plan π, and a_s is the starting action of plan π, executed immediately after the recovery procedure.[10] In general situation, the robot can receive different demonstrations, given a failure of an action a_i for condition ϕ_i; this generates a data set of user demonstrations D_{a_i,ϕ_i}. From these initial element the goal is build a robust plan π^* that consent the robot to overcome the failure condition ϕ_i in execution of a_i, according to the demonstrations, and continuing the execution of π.
The solution of this problem is divided in three phases:
1) Obtain demonstrations from not-expert user. In this case human-robot interaction techniques are applied (for example, through use of a tablet mounted on the robot) for: labelling some specific states of the plan with a semantic representation, which allow the robot to communicate what it is doing: its current state, future steps of the plan, final goal, and eventually a list of available actions; sending to robot a sequence of actions [11] for go over the current failure situation and to bring it to a step in the nominal plan π.
2) Learning a recovery procedure. The learning procedure is based on a transformation of the data set D_{a_i,ϕ_i} into a controllable Decision Process [6], and on the application of Reinforcement Learning[12] to it. The aim of this phase is learn a proper policy p^* that maximizes the reward on the Decision Process generated from the user demonstrations.
3) Generate a robust plan. Integrating in the nominal plan π the learned recovery policy p^*, through the use of plan representation formalism, Petri Net Plans [8], so as to obtain obtain a new plan π^*.
These three phases can be replay over time, producing a continuous learning system.

Del Duchetto et al. [2] applied a similar approach to improve the robustness of navigation plan for autonomous mobile robots, in long-term scenarios. The basic methodology is developed for two purposes: 1) Detecting a navigation failure in real-time; achieved by Gaussian Process(GP) binary classifier [12]. 2) Learning local recovery policy for the failure situation; through the use of Gaussian Process(GP) regressor [12].
When GP classifier identify a situation that robot cannot manage, the navigation is interrupted. The human is called to confirm the pertinence of the detection. If human

[10] Note: sequence of actions in $p = \langle \alpha_1, ..., \alpha_r \rangle$ are performed by the robot, but is not require them to be modelled in its knowledge based.

[11] Note: in this phase, human-robot interaction interface allows the robot to collect a various of pairs (p, a_s) and increase the size of data set D_{a_i,ϕ_i}

[12] Note: the rewards can be obtain in two ways: automatically, achieving the goal with a proper execution of a recovery procedure; from users, interrupting the current recovery procedure giving a negative feedback.

user rejects the detected state as a failure: training set and the classification model are updated, and the global navigation plan is resumed. Otherwise, if it is confirmed, a Bayesian likelihood method (with a fixed threshold) [13] is used to evaluate the goodness-of-fit for the regression model. In positive case, the GP regression model is used to execute the learned recovery policy. In negative case human intervention is required: human user assume control of the robot to demonstrate how to recover from the failure situation. This demonstration is used to train a GP regressor (improving model performance). Put more simply, is applied an active learning approach, where the system automatically evaluates the relevance of failure states to already seen ones, and demands new demonstrations when necessary.
The empirical results on two different failure scenarios(i.e., robot attempts to pass through a narrow passage between two obstacles, the robot needs to turn a corner in order to reach a goal) show that given 40 failure state observations, the true positive rate of the failure detection model exceeds 90%, ending with successful recovery actions in more than 90% of all detected cases.

Chapter 3

Software and Tools

3.1 Docker : containerization platform

Docker is a software platform (open-source) that allows to develop, test and deploy applications guaranteeing portability and self-sufficiency basically in all computing environments. This paradigm has the advantage of simplifying, automating and accelerating application development (for instance, a programmer will focus mainly on the application code, without spending a lot of his time on the runtime environment configuration; and in case of a modification or upgrade of the starting configuration leads to errors or system conflicts, restoring the previous situation is almost immediate; this last aspect encourages and allows easier experimentation), and the lifecycle of implementations. Practically Docker collects software into independent and standardized units that offer everything needed for their proper execution, including libraries, run-time, and system tools.

The fundamental Docker concepts are described by "images" and "containers". An image is a package with all the necessary dependencies and information (code, run-time, drivers, tools, scripts, libraries, and more) to create a container in which a certain application can be executable. Each image is defined by a Dockerfile, a configuration file that contains all the commands to assemble the image. Docker Hub[1] is a cloud-based registry service that shares repositories of docker images created by the community.

A container is a runnable instance of a image: it is the independent and standardized units that contain everything (code, run-time, system tools, libraries...) needed to run an application, from one computing environment to another. Note: containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware : a container runs on the host operating system kernel. This make it more portable, lighter weight and resource-efficient than virtual machines.

The initial part of this thesis was dedicated to creation of a docker container in which to run mainly two robotic software: ROS and Gazebo simulator. One of the main drawbacks encountered, (and more generally related to Docker technology), is the fact that Docker was designed as a solution for deploying console-based applications, that don't require a graphical interface. This was one of the crucial points to be addressed at the beginning of this thesis work (which took some time to overcome), where a graphical user interface software (Gazebo) was needed for the simulation of the robot.
Running graphical applications inside Docker containers requires the study of some "precautions" (such as X11 video forwarding) that must be evaluated. In Appendix A.1, the Linux configuration used to launch a docker container created for this thesis work, containing a graphical simulation of the UR5 robot in ROS - Gazebo environment.

Also in the Appendix A.2, are present the specifications of the configuration system of the container used.

[1] https://hub.docker.com/

3.2 ROS : framework for the robot applications

This thesis work was developed through the robotics middleware suite ROS; according to what the authors defined[25] :

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

In this framework, the general structure of an application can be seen as a network of independent executable programs, called nodes, that performs the elaboration of data; which, interconnected with each other into a graph, exchange messages within channel, called topic, if the mode is asynchronous, otherwise if the procedure is synchronous(request/response) they are said: services. More details will be explained below.

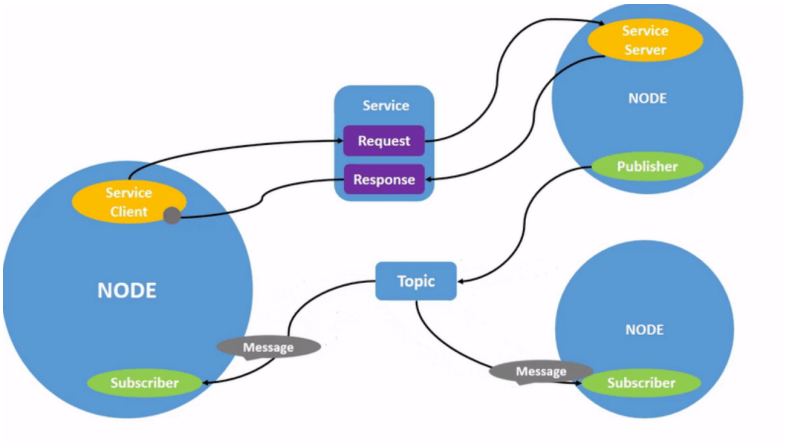

Figure 3.1. Simple scheme of node graph architecture in ROS [26]

Structurally ROS develops on three levels of concepts:
Filesystem level, it includes all ROS resources on local disk :

- Packages : the main unit for organizing software in ROS, they contain ROS runtime processes nodes, a ROS-dependent library, configuration files, datasets and all the other files that are used by the system for execution - concretely it is represented by a folder, within which it includes file and some sub-folders to manage the system (e.g. package.xml, CMakeLists.txt:, folder scripts, config, launch and more)
- Metapackages : specialized structures for represent a group of related other packages.
- Package Manifests : the file package.xml, provide in the form of metadata characteristics and information regarding a package: its name, version, description, license, dependencies, ...
- Message Type : define the data structures for messages sent in ROS. Stored in folder /msg, each file, with extension .msg, indicates a different type of

message. Inside the files, each line represents a data field, structured in a type (int, float, bool, string...) and a name (how a data value is referenced in the target language), separated by a space.

- Service types: define the request and response data structures for services in ROS. Stored in folder /srv, file extension .srv; they have a structure similar to message type (i.e. each line represents a data field), with a difference, the file is divided into two parts (separated by '—'): first part is relative to the request (constant and field dependent), while the other related to the response.

Computation graph level, represents the peer-to-peer network system that is used for communication and processing data. The basic element are:

- Nodes: are processes that perform computation inside ROS network. A complex robot system include many nodes, usually each node is related to specific functionality : e.g. one node performs localization, one node performs path planning, one node provides a graphical view of system, ect... ROS discourages the creation of nodes that perform multiple functions, so as to make the network more manageable, intelligible, re-usable and tolerant to errors. The nodes are usually developed in C++ or Python, with the use of a ROS client library, such as roscpp or rospy.

- Master: supplies naming and registration services to the nodes of ROS network. It also tracks publishers and subscribers to each topics, as well as services. I.e. it allows a single node to locate one another and communicate peer-to-peer.[2] It is the core of network; without it, network is not be able to work.

- Messages: the communication between nodes takes place via messages exchange. This data can support standard primitive types (integer, floating, boolean, etc.) and arrays of primitive types.

- Topic: are buses unidirectional that consent the exchange of messages between nodes in asynchronous mode. Topics have anonymous publish/subscribe semantics : a node that wants to share information publish messages on a certain topic, another node that is interested (wants to receive) in that information will subscribe to the same topic. Anonymous means that publishers only knows it is publishing to a topic, and a subscriber only knows it is subscribing to a topic (i.e. nodes are not aware of who they are communicating with). This consent to decouple the production of information from its consumption. Each topic has a strongly typed message. All publishers and subscribers on same topic have to use the message type associated with it.

- Services: represent the synchronous means of communication between nodes, based on the request / reply mode : i.e. a pair of message structures: one for the request and one for the reply. In this way, in the network will be defined nodes that act as service provider and client; a node client sends a requests message, and wait until it receives a response from node provider.

Community level, ROS provides developers with resources for a constant exchange of information and software. These include: a forum for documenting information about ROS (called The ROS Wiki), a Q/A site for answering (ROS Answers), a blog, a collections of versioned software ready for installation and constantly updated (ROS Distributions), a federated network of code repositories (ROS Repositories).

[2]Note: nodes connect to other nodes directly; the Master only provides lookup information.

The core of the data flow ROS system of this thesis work is visualized in the following graph:

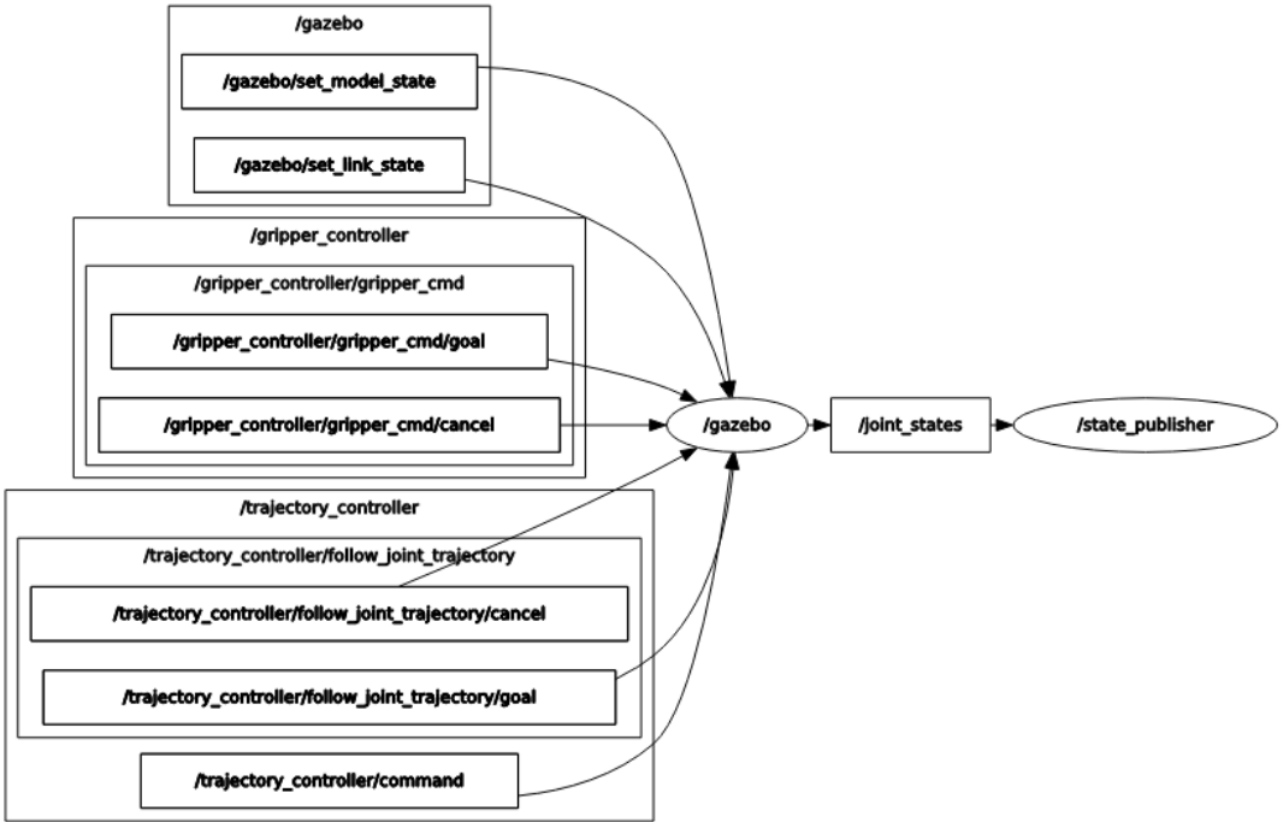

Figure 3.2. The Graph shows structure of the thesis's ROS network core. Obtained installing and using the tool: `rqt_graph`

The graph above shows the relationships between ROS nodes core; the additional nodes used in carrying out the experimentation will be described in detail later. The topics are mentioned in a rectangle and nodes in circles. The arrows indicate the direction of the message between: the node that publishes on the topic and the node that subscribe. Note to make the diagram more intelligible, have been omitted the topic without a subscriber and all topics used for debugging and logging message (like `/clock` topic).

Gazebo simulator(3.3) starts as a ROS node called `/gazebo`. It receives data from:

- `/gazebo` subscribed topics: ROS API interface that sets the state (pose/twist) of a link and a model.
- `/gripper_controller` topics used to control the Robotiq 85 gripper
- `/trajectory_controller` topics used to control the 6 joints of UR5

And publish data on `/joint_state` topics. The messages (type: `sensor_msgs/JointState`)on this topic report the current state of gripper and 6 joints of UR5 in this structure:

```
Header
  uint32 seq
  time stamp
  string frame_id

string[] name
float64[] position
float64[] velocity
float64[] effort
```

Note: All arrays (name, position,..) in this message have the same size: 7 information (1 for gripper + 6 for UR5 joint).

At the end, the node `/state_publisher` receives the type of information described above, that help you to broadcast the state of robot via tf [3]

* * *

The main advantages given by the choice of ROS for a projects are:

- Hardware abstraction and Modularity: it facilitates the development of a robot system and lends to reuse and sharing of node-code in more than one projects (as long as you respect the dependencies); in more, if only one node crashes, the system can still work. This aspect combined with its inter-platform operability (means: we can write some nodes in C++ and other nodes in Python or Java) makes ROS extremely stable and flexible.
- It was designed to be used within distributed systems: this allows the communication between the various processes running, regardless of whether they reside on the same machine, or spread across different machines.
- It is open source and it is supported by a large active community. It is becoming a default format in commercial robots as well [4].
- The wide range of tools for debugging, visualizing, performing simulation, drivers and interface packages of various sensors (e.g. Velodyne-LIDAR, Laser scanners, Kinect and so on) and actuators supported.

[3] tf is a package that lets the user keep track of multiple coordinate frames over time

[4] Nearly 55% of commercial robots shipped by 2024 will have at least one ROS package on board. Source of information: ABI Research; https://www.abiresearch.com/blogs/2019/06/10/ros-rise/

3.3 Gazebo : simulation environment

A simulator can recreate many aspects, but not all, of a robot and the real world in which it interacts; such as: robot geometry, obstacles, movement, sensor readings and sensor feedback process,etc. This basically allows to reduce the high costs of design, experimentation and debug code, which would entail working on hardware in real world situation.

Gazebo[5] was chosen in order to make dynamics simulations of a ROS node system without need to have the physical hardware of the robot [6]. It is an open-source 3D robotics simulator that consent to create realistic simulations of robots in its work environment (putting into account: force of gravity, collisions, friction, lights, ect ...)[7]; quickly test algorithms in it, and, train AI system using realistic scenarios. In Gazebo it is also possible to generate data (optionally with noise) from different sensors: laser range finders, RGB cameras, kinect sensors, contact sensors, and more; allowing to control and modify the actions of the robot model based on the data generated.

The main advantage of using Gazebo is its modular structure: it is possible to insert (o remove) new functionalities in the simulation model through the logic of plugins: a plugin is a chunk of code, compiled as a shared library, attached to a world or model, and loaded when Gazebo is launched. Gazebo supports several plugin types: World, Model, Sensor, System, Visual, GUI. They are selected based on the desired functionality. E.g., use a Model plugin to have the control of joints, and state of a model; or use a Sensor plugin to obtain sensor information and control sensor properties. This modular structure simplifies and makes very flexible the use and creation of new models of robots.

For the thesis work it was necessary to recreate a simulated robot model and a simulated working environment in which to experiment our purposes; for the reasons expressed above, Gazebo simulator was considered the most suitable. Having received as input the information about the real structure of Development and Research laboratory of Procter & Gamble -Belgium in which the experimentation takes place, see the two pictures below:

[5] http://gazebosim.org/

[6] Note Gazebo is independent from ROS and is installed as a standalone package. For properly interface Gazebo and ROS see in the Appendix A.2

[7] Note: inside the simulator, each object has its own physical and visual characteristics, in order to make the simulation as realistic as possible

Figure 3.3. Photo of the laboratory. This photo has been granted by Fabio Zonfrilli, Research and Development laboratories of Procter & Gamble Services Company - Belgium. Tools in picture: UR5 (1), drawers (2) containing the pouchs, Mark10 Tigthness (3), Mark10 Strength (4), vacuum gripper (5), digital scale - missing in this photo but but placed at the point (6), camera attached to the wrist (7). Note: to ensure trade secrecy, certain specific information and few tools have been omitted.

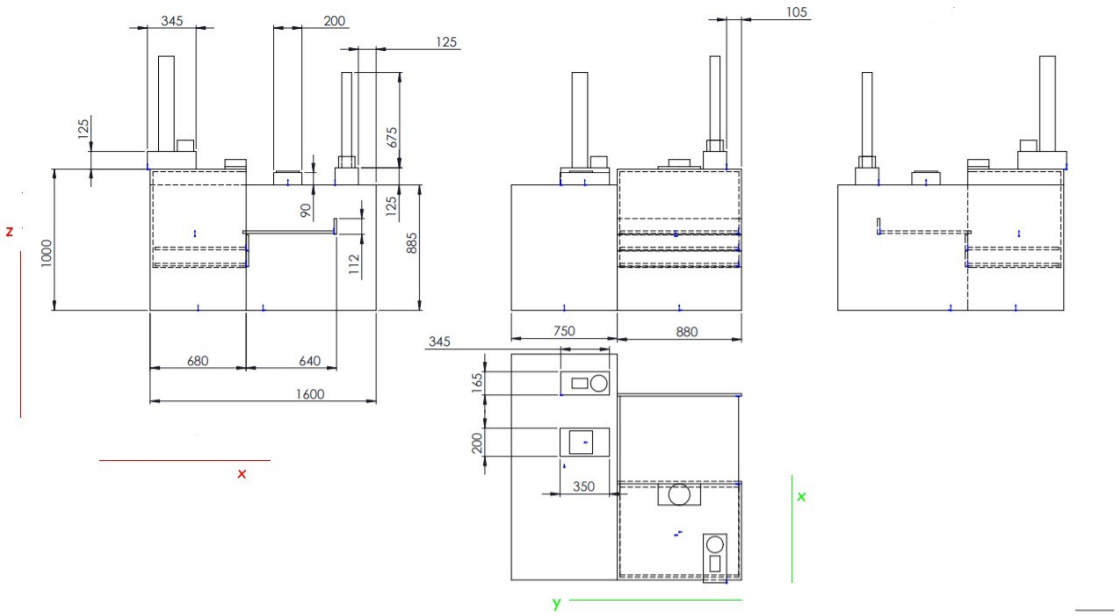

Figure 3.4. Scheme of the workstation with the relative measures

I have reconstructed the laboratory environment in the simulated space of Gazebo,

with all the relative specifications: from the position of the tools to their structural dimensions, passing through the simulation of gravitational effects. The result can be see below:

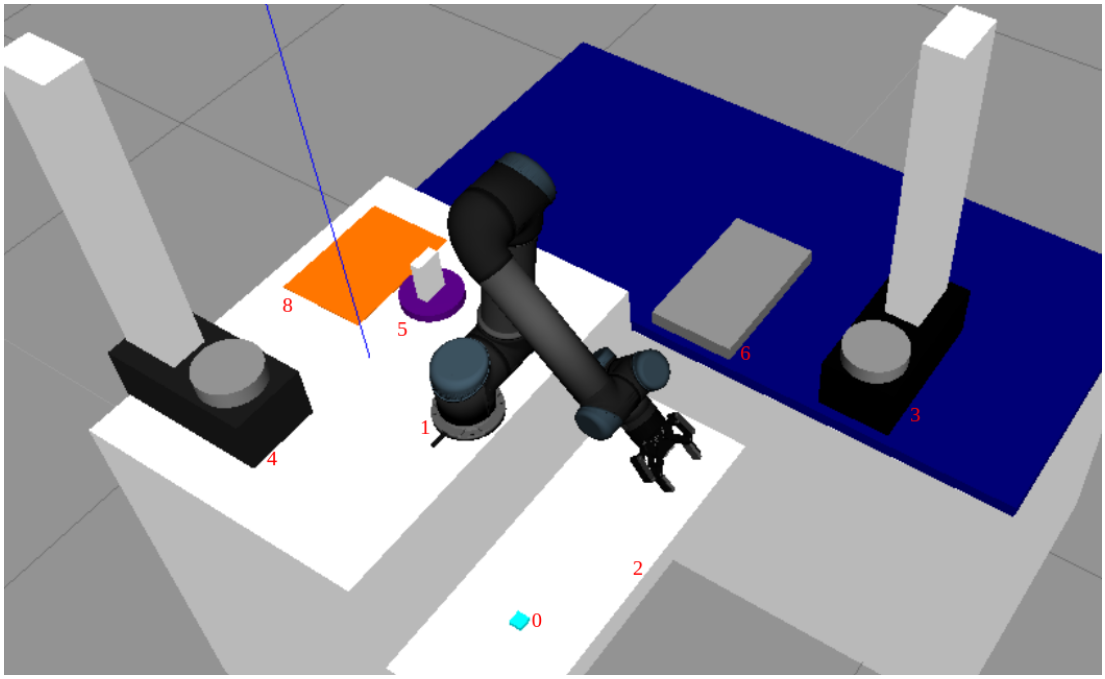

Figure 3.5. Picture of the Gazebo simulation environment. Notation of simulated tools are the same of picture 3.3: UR5 (1), open drawer (2) containing the pouch (0) in turquesa, Mark10 Tigthness (3), Mark10 Strength (4), vacuum gripper (5), digital scale (6), plastic sheet (8) in orange.

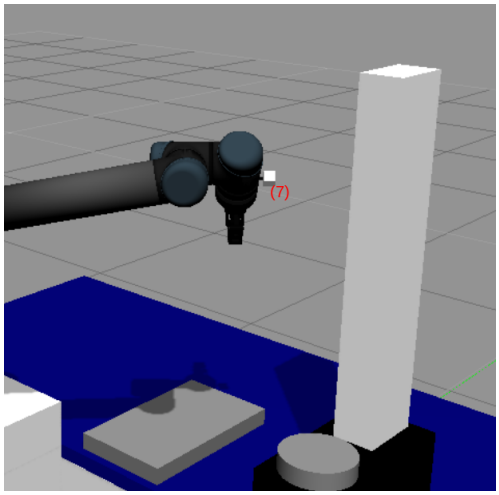

Figure 3.6. Picture of the Gazebo simulation environment. The white box represents the camera attached to the wrist (7).

For the define the “world” present in the laboratory, by that I mean all you can see in Figure 3.5 except for the robot and sensor (for which a URDF was adopted [8]), was used SDF (Simulation Description Format) file. This format is structured as follows: at the beginning the general aspects of the world in which all objects will stay are defined: physics (gravity vector), global lights, ground plane,... Each object, tagged `<model>`, is basically composed of a serial link, for which different aspects are defined: the position in world coordinate, the static (if true mean that object is immovable), the collision aspect (i.e. it encapsulates a geometry that is used for collision checking), the visualize part, the inertial element (i.e. the dynamic properties of the link, such as mass and rotational inertia matrix). For instance, a part of SDF code used for the construction of the simulation environment will be shown in Appendix A.4.

About the physical structure of camera as "extra link" attached to the wrist; its size and coordinates of the position are defined in the file `ur5.urdf.xacro` (see the Section 3.4). In particular the piece of code that determines its position on wrist is:

```
<joint name="camera_joint" type="fixed">
  <axis xyz="0 0.7 0" />
  <origin xyz="0.08 0.0 0.045" rpy="0 0 0"/>
  <parent link="ee_link"/>
  <child link="camera_link"/>
</joint>
```

About the plugin that allow the camera functionality and publishes the image to a ROS message : the camera sensor model used is already implemented in the package `ros-kinetic-gazebo-ros-pkgs`. It is loaded by inserting it into the file : `common.gazebo.xacro` [9]. In details, in the first part of the file, the camera specifications are defined, as:
the number of frame-image per second taken within Gazebo:

```
<update_rate>30.0</update_rate>
```

set the resolution of the image:

```
</image>
   <width>800</width>
   <height>800</height>
   <format>R8G8B8</format>
 </image>
```

set the parameters "near" and "far", that give an upper and lower bound to the distance in which the cameras can see objects in the simulation:

```
<clip>
  <near>0.07</near>
  <far>300</far>
</clip>
```

The camera will output images with 800x800 resolution, with a 24-bit RGB pixel format(8 bits per channel), at 30 frame per second. In the second part of file are described the plugin definition:

```
<plugin name="camera_controller" filename="libgazebo_ros_camera.so">
```

[8] Note: basically URDF and SDF are similar for XML and tags, this it will be clear in next section

[9] `/root/catkin_ws/src/ur5/ur5_description/urdf/` path to locate the file in thesis Docker container

that describes where the actual `gazebo-ros-camera.cpp` file is linked to, as a shared objec.

```
<cameraName>ur5/usbcam</cameraName>
<imageTopicName>image_raw</imageTopicName>
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
```

Here has been defined the camera name, and the two ROS topics on which the data-image will be published: image topic and the camera info topic.

For testing the plugin, and put out what the camera is filming, assumed that Gazebo is running, the following command is used:

```
rosrun image_view image_view image:=/ur5/usbcam/image_raw
```

The result is shown in the next picture:

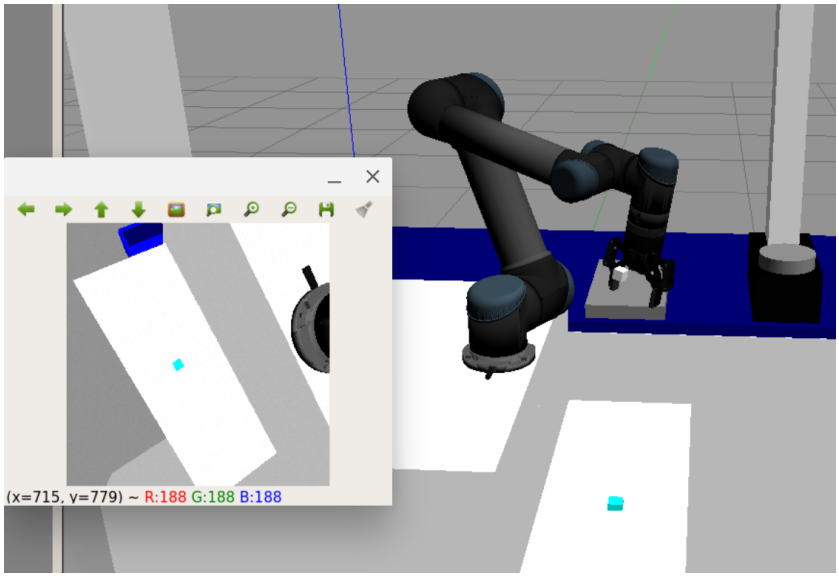

Figure 3.7. Picture of the Gazebo simulation environment. The small panel shows the output of camera attached to the wrist: it is pointing to pouch in open drawer. You can also see the corner of bin (in blue) on the ground.

To launch everything, was create a file `.launch` [10]: in which are setting arguments (as: gui, debug, paused...) to control the behavior of Gazebo, export env variable to find the location of Gazebo resources, resume the logic in `empty world.world` (already defined in the Gazebo), load the URDF robot model into the ROS parameter server, and spawn the robot into Gazebo. To launch the file is used the following command:

```
roslaunch ur5_gazebo ur5_setup.launch
```

Further information on the tools, will be described in the next sessions; this section aims only to present the work done to build the simulator.

[10] `/root/catkin_ws/src/ur5/ur5_gazebo/launch/` path to locate the file in thesis Docker container

3.4 Unified Robotic Description Format (URDF) : robot model

The Unified Robotic Description Format (URDF) is an XML file format used in ROS to define a robot model and sensors. The format provides a description of the robot using a tree-like structure whose main nodes are the `links` , used to describe the rigid components of the robot, connected to each other by `joints` [11].

At each `link` is assigned a name, and the following specifications :

- the graphic aspect (tagged `<visual>`). The geometry of the links is described by graphic primitives (cylinder, box, sphere)included in format, or by using a mesh (usually in format .dae).
- the characteristics relating to the physics of the link: mass and inertia (tagged `<inertial>`)
- the collision properties of a link(tagged `<collision>`. These can be different from the visual properties: a simpler collision models is used to reduce computation time.

Each `joint` is specified by a name and a type : revolute, continuous, prismatic, fixed, floating, and planar. This consent to describe how the links are connected to each other through a Parent-Child relationship.

Below, a part of the code of the URDF model[12] used in this thesis work is shown. The name of the file in which it is contained is: `ur5.urdf.xacro` [13]. This describes two links: the forearm and the first wrist, properly: link 3 and link 4 in figure 3.9; and the joint between them, which define their relationship parent-child:

```
<link name="${prefix}forearm_link">
 <visual>
   <geometry>
     <mesh filename="package://ur5_description/
     meshes/ur5/visual/forearm.dae" />
   </geometry>
   <material name="LightGrey">
     <color rgba="0.7 0.7 0.7 1.0"/>
   </material>
 </visual>
 <collision>
   <geometry>
     <mesh filename="package://ur5_description/
     meshes/ur5/collision/forearm.stl" />
   </geometry>
 </collision>
 <xacro:cylinder_inertial radius="0.06" length="0.5"
 mass="${forearm_mass}">
   <origin xyz="0.0 0.0 0.25" rpy="0 0 0" />
 </xacro:cylinder_inertial>
</link>
```

[11] Note: Flexible links can't be represented using URDF.

[12] Based on the official code distributed by company Universal Robots: https://github.com/ros-industrial/universal_robot

[13] `/root/catkin_ws/src/ur5/ur5_description/urdf/` path to locate the file in thesis Docker container

```
<joint name="${prefix}wrist_1_joint" type="revolute">
  <parent link="${prefix}forearm_link" />
  <child link = "${prefix}wrist_1_link" />
  <origin xyz="0.0 0.0 ${forearm_length}" rpy="0.0
  ${pi / 2.0} 0.0" />
  <axis xyz="0 1 0" />
  <xacro:unless value="${joint_limited}">
    <limit lower="${-2.0 * pi}" upper="${2.0 * pi}"
    effort="28.0" velocity="3.2"/>
  </xacro:unless>
  <xacro:if value="${joint_limited}">
    <limit lower="${-pi}" upper="${pi}" effort="28.0"
    velocity="3.2"/>
  </xacro:if>
  <dynamics damping="0.0" friction="0.0"/>
</joint>

<link name="${prefix}wrist_1_link">
  <visual>
    <geometry>
      <mesh filename="package://ur5_description/meshes/
      ur5/visual/wrist1.dae" />
    </geometry>
    <material name="LightGrey">
      <color rgba="0.7 0.7 0.7 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <mesh filename="package://ur5_description/meshes/
      ur5/collision/wrist1.stl" />
    </geometry>
  </collision>
  <xacro:cylinder_inertial radius="0.6" length="0.12"
  mass="${wrist_1_mass}">
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
  </xacro:cylinder_inertial>
</link>
```

During the experimentation work, we are interested in keeping the model permanently on the table, and not falling down under the effects of gravity for certain maneuvers. For obtain this result, i used a "world" link and a joint that fixes it to the base of your model:

```
<link name="world" />

<joint name="world_joint" type="fixed">
  <parent link="world" />
  <child link = "base_link" />
  <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
</joint>
```

`"fixed"` means: all degrees of freedom of a joint are blocked,this allow to connect two links without them being able to move relative to each other. The same approach was used to permanently attached the camera model to wrist :

```
<joint name="camera_joint" type="fixed">
  <axis xyz="0 1 0" />
  <origin xyz="-0.02 0.0 0.055" rpy="0 0 0"/>
```

```
    <parent link="${prefix}wrist_3_link" />
    <child link="camera_link"/>
  </joint>
```

3.5 Universal Robots 5 (UR5)

Universal Robots is a Danish company that produces collaborative industrial robots (cobot): i.e. a robot designed to physically interact with humans in a workspace. UR5 is collaborative robotic arm, launched on the European market in 2008. Some specifics: about 20 kg of weight for a footprint of 149 mm. Performance: 5 kg of payload, 360° movement on six axes, a working radius of 850 mm. Given its versatility it is used in various contexts: from picking to assembly, from quality control to machine tending.

Figure 3.8. Photo of the robot Universal Robot 5 [33]

For more details I suggest to visit the site: [33].

Follows a diagram of UR5 robot parameters that using the Denavit-Hartenberg method.

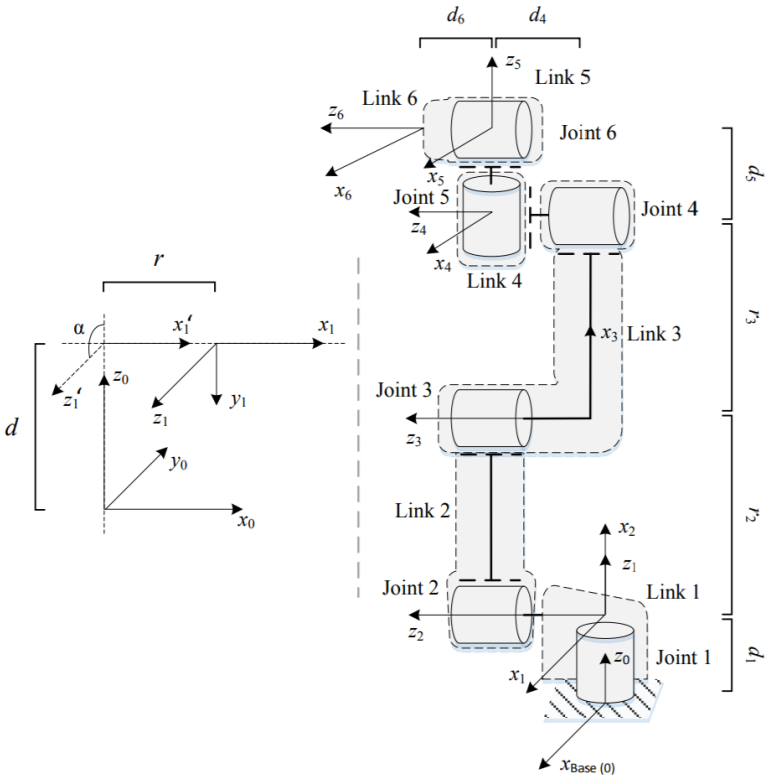

Figure 3.9. UR5 robot diagram by D-H [39]

Kinematics	r_j	d_j	$alpha_j$	Dynamics	Mass (kg.)
Joint 1	0	0.089159	π/2	Link 1	3.7
Joint 2	0.425	0	0	Link 2	8.393
Joint 3	0.39225	0	0	Link 3	2.33
Joint 4	0	0.10915	π/2	Link 4	1.219
Joint 5	0	0.09465	- π/2	Link 5	1.219
Joint 6	0	0.0823	0	Link 6	0.1879

Figure 3.10. UR5 robot kinematic and dynamic parameters [39]

Note:
During the thesis work it was necessary to work in inverse kinematics: given the values in cartesian coordinates (x, y, z, referred to the space-frame Gazebo) of the gripper center, move the UR5 adequately, attributing the right values to its six joints. To achieve this, Ryan Keating's documentation was used [37].

3.6 Robotiq 2F-85

In laboratory, the gripper connect to Universal Robot 5 is the Robotiq 2F-85. Some specification: Stroke: 85 mm, Grip Force: 20 to 235 N, Closing speed: 20 to 150 mm/s.

Figure 3.11. Photo of Robotiq 2F-85 on UR5

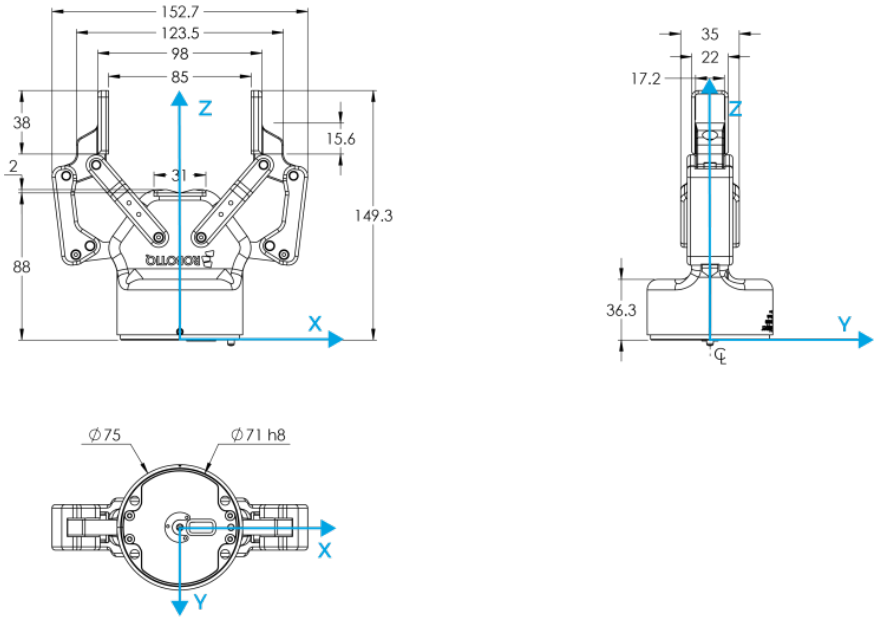

Figure 3.12. Robotiq 2F-85 technical drawing

For more details I suggest to visit the site: [34]

3.7 Petri Net Plans (PNP)

Petri Net Plans is a framework for high level description of complex plans, based on graphical modeling language Petri nets [7]. The reason why it is interesting is well explained by the words of its creators:

PNPs are inspired to languages for reasoning about actions, yet they are more expressive than most of them, offering a full fledged set of operators for dealing with non-instantaneous actions, sensing actions, action failures, concurrent actions and cooperation in a multi agent context. PNPs include also choice operators used for non-deterministic execution and for learning in the plan space through a Reinforcement Learning algorithm. ... PNPs are more expressive than Finite State Machines and allow for automatic plan analysis, which can provide formal guarantees on the performance of the plans. Execution of PNPs is extremely efficient and allows the design of real-time pro-active and reactive behaviors. [14]

Petri Net Plans can be explained starting from Petri Net:

Petri Net is a a directed, weighted, bipartite graph that has two types of nodes: places and transitions, connected by directed weighted arcs. Place can contain any number (including zero) of tokens, the number of tokens over the places will represent a configuration of the net called a marking and denotes the state of the system. Transitions can consume or produce tokens from places according to the rules defining the dynamic behavior of the Petri net: the firing rule. A transition of a Petri net can "fire" if it is enabled, i.e. there are sufficient tokens in all places connected to it as inputs; when the transition fires, it consumes the required input tokens, and creates tokens in its output places. Transitions represent the events modeled by the system. Arcs run from a place to a transition or vice versa, never between places or between transitions.

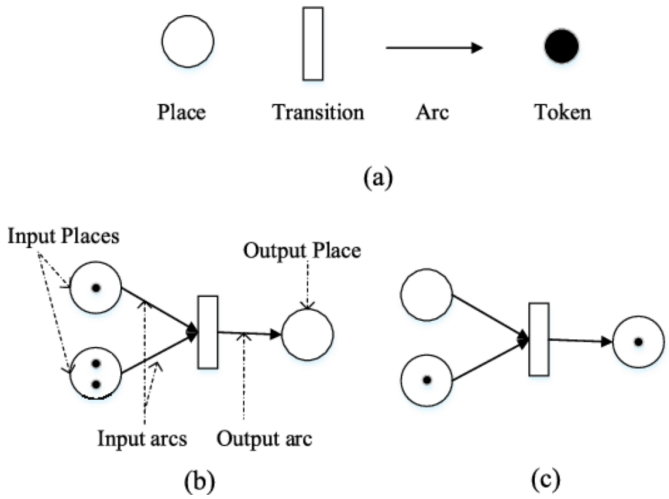

Figure 3.13. Petri net graph: (a) basic elements (b) before transition firing (c) after transition firing. [10]

[14] https://sites.google.com/a/dis.uniroma1.it/petri-net-plans/

With this notions in mind, is possible to think of a Petri Net Plan as a tuple : $\langle P, T, F, W, M_0, G \rangle$:

- $P = \{p_1, p_2, p_3, ..., p_m\}$ is a finite set of places. Places p_i represent the execution phases of actions; each action is described by a place corresponding to its initiation, one to its execution, and one to its termination;
- $T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions. Transitions t_i represent events and are grouped in different categories: action starting transitions, action terminating transitions, action interrupts and control transitions. Transitions can be labeled with conditions that control their firing.
- $F \subseteq (P \times T) \cup (T \times P)$is a set of edges.
- $W : F \to 1$ is weight function for edges. Denoting the weight of the edge from f_s to f_d with $w(f_s, f_d)$; for each pair f_s, $f_d \in F$ is $w(f_s, f_d) = 1$
- $M_0 : P \to \{0, 1\}$ is the initial marking (representing a description of the initial state of the agent.)
- G is the set of desired goal markings for the agent (and is a proper subset of the possible markings that the PNP can reach.)

To deepen the topic of Petri Net Plans, I suggest reading the paper [8]. For the purposes of this thesis, Petri Net Plan provides high level descriptions of complex actions interaction to deal with failure situations (essentially, provide a mechanism for checking and interrupting the the execution of the nominal plan in case of failures). For this reason, I consider it appropriate to present the plan syntax for PNP generation used in this thesis work; necessary to understand the generated plans for the experimentation phase, and presented as a conclusion to this fifth chapter. The basic points to keep in mind are the following:

- The plan, thought of as a sequence of actions, is modeled, in a file `.plan`,in this way:

```
action1;
action2;
action3
```

  While for addition of conditions:

```
< conditionK?
    action1; action2; action3 :
 (not conditionK)?
    action4; action5
>
```

  That means, if conditionK is true, do the sequence of action: action1; action2; action3; if false do action4; action5.

  The command `LABEL` is used to identify a position in the plan, while the command `GOTO` serves to link to that position:

```
action1;
LABEL_ABC;
action2;
< conditionK?
   action3 :
(not conditionK)?
   action4; GOTO_LABEL_ABC
>
```

That means, if conditionK is false, after the execution of action4, the plan continues by starting the action2.

Any part of the file beginning with `#` is ignored; it is to be considered an external comment.

- Actions can have parameters as input: the character `_` is used to distinguish names of the actions from their parameters.

```
goto_placeXYZ;
say_hello;
wait_10
```

In this example the agent will first go to the place indicated by the parameter: `placeXYZ`, then will say the content of the parameter `hello`, and wait for 10 seconds.

- Execution Rules (ER): are statements that allows to define additional conditions to a conditional plan. This condition are evaluated in real time during execution of plan: the checks occur throughout the execution of an action, at a frequency of 10 Hz. The use of ER simplifies a lot the addition of special cases that otherwise would need to be encoded in the planning domain.

  Execution rules have the following form:

```
*if* condition *during* action *do* something
```

  This "something" in the end, which must be executed if a certain condition occurs during a certain action, can be:

  `skip_action`: the current action is skipped.

  `restart_action`: the current action is restarted.

  `GOTO_LABEL_ABC`: the plan continues from `LABEL_ABC`.

  `restart_plan`: the all plan is restarted.

  `goal`: the plan terminates with success.

  `fail_plan`: plan terminates with failure.

  `anotherplan` that is a another plan (i.e., just an action, or a sequence of actions with or without conditions) terminating with one of the command above listed.

  ER can be stored in a separate file from the plan, with the extension `.er`; in this case ER are applied to each action of the plan mentioned in the ER statement. Or, can be written in the file of plan `.plan` to be applied to a specific instance of an action:

```
goto_placeXYZ;
say_hello;
goto_placeXYZ; ! *if* conditionK *do* something !
wait_10
```

In case above, ER will be applied only to the action that is defined immediately before the ER statement (second instance of `goto_placeXYZ)`. Note: in this case the syntax of ER is simplified since it is not necessary to specify the action.

For more details on the syntax of the plan I suggest to read: [11].

3.8 Plan Execution Interface (PLEXI)

PLEXI concept is explained comprehensively by the words of its author [38]:

A Plan Execution Monitor (PEM) aims at orchestrating the execution of actions and monitor the values of fluents (or predicates), following a plan generated to achieve a given goal. The implementation of the interface between a PEM and action/fluent implementations may be non-trivial and it is usually achieved through specific coupling PEM and action/fluent implementations. Existing examples are ... PetriNetPlans ActionServers. Plan Execution Interface (PLEXI) is a layer for increasing interoperability between a PEM and the implementation of actions and fluents in a complex system (e.g., a robotic application). PLEXI provides an interaction protocol between PEM and action/fluent implementations, allowing for decoupling all these components. With PLEXI, action/fluent implementations will not depend on the specific PEM. There is no need to import libraries or include code from the PEM, there is no need to know which PEM will orchestrate the implemented actions/fluents.

Note to the reader: In this context, the terms conditions, fluents, and predicates are used as synonyms.

The actions, essentially are a ROS node, in my case in Python, allow to move the robotic arm (`goto`), open and close the gripper (`movegripper`), interact with the user (`say`, `waitfor`), wait for the execution of a certain external operation to be finished (`wait`), and set the fluents value (`sense`), i.e. conditions that change their status over time, and which determine the progress of the plan. When a plan is started, for each action the PEM publishes on a ROS topic a String message command: start (an action thread is created and then launched), interrupt (action thread is paused), resume (action thread is resumed). For more detail see section 5.4. For an example of action see appendix A.7.

The fluents, as mentioned above, are conditions that determine the progress of the plan. Also them written in Python, they receive information (i.e. observation) from the environment as input, process it, giving as output: condition to 1 (true), 0 (false), or -1 (neither true nor false, unknown, probably something went wrong). In this thesis, `presentpouch` should allow to detect the presence of the pouch within the camera frame; `grasppouch` if the pouch is grabbed by the gripper , `Ask` to receive replies (yes / no) from the user.

For more information i suggest reading: [38].

Chapter 4

AI Planning Design

4.1 Illustration of the nominal plan

The objective of this quality control activity is to analyze and evaluate the performance of a P&G laundry pouch (Aka pod), through the use of a robotic arm (described in the previous section) and a series of tools: digital scale, Mark10 Tigthness(tightness test stand), Mark10 Strength (strength test stand).

The nominal plan is a sequence of actions that when successfully executed make the task fully accomplished. For the rest of the session i will describe briefly the phases of the this plan in order to explain to the reader the expected behavior of the robot, assuming that everything goes right, and no checks and recovery procedures are required. To describe the procedures of the nominal plan, I found it helpful to divide the it into 5 stages:

Start up procedure

1. Activate the robot.
2. Give the UR5 the coordinates of the joints to "see" the pouch; i.e. place the robot wrist camera on top of the tray so that its contents are fully visible. What I mean exactly is shown in the figure below.

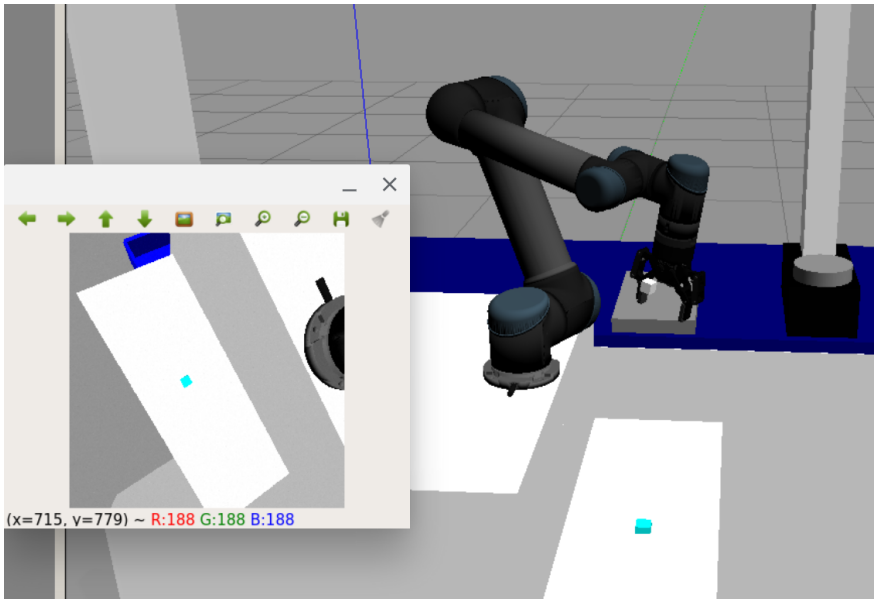

Figure 4.1. Vision pose of UR5

Weighing procedure

1. Active vision procedure to get the exact 3D location coordinate of pouch in tray.
2. Pick pouch, i.e. move robot arm joints until reach the position of the pouch, and close the gripper to grab it.
3. Move robot grasping pouch on the center of scale.
4. Leave pouch on scale and move away.
5. Wait for weight procedure measurement (including the record of the weight value)

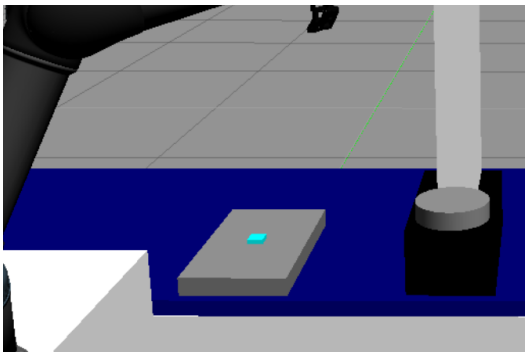

Figure 4.2. Weighing final pose

Tightness procedure

1. Pick pouch from scale.
2. Move robot grasping pouch on the bottom plate of Mark10 Tigthness.
3. Leave pouch on Mark10 Tigthness and move away.
4. Wait for the tightness procedure measurement to complete (i.e. the height of the pouch is measured at a compression of 10N).

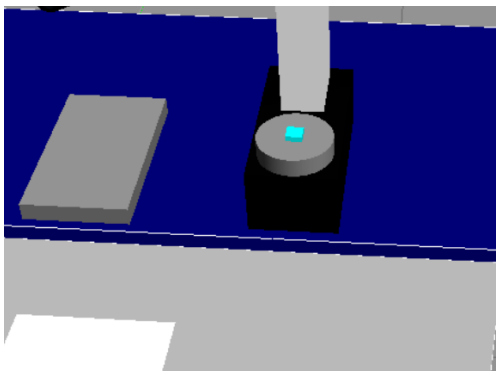

Figure 4.3. Tightness final pose

Strength procedure

1. Pick the vacuum gripper. Note: the gripper needs to be free; i.e. not holding a pouch, this has been left on Mark10 Tigthness.
2. Place the vacuum gripper on the top plastic sheet.
3. Activate vacuum gripper.
4. Place the plastic sheet (think of this as the first layer in whick the pouch will be crushed) on the center of Mark10 Strength instrument.
5. Pick the vacuum gripper. Note : The gripper needs to be free; i.e. not holding a pouch, this has been left on Mark10 Tigthness.
6. De-activate vacuum gripper
7. Put back the vacuum gripper on its holder.
8. Pick the pouch from Mark10 Tigthness.
9. Place the pouch on the plastic sheet placed on Mark10 Strength.
10. Repeat steps 2 to 7 to place the second plastic sheet (second layer).
11. Start the strength procedure (approx. 1 minute); in which the pouch will be squeezed between the two sheets, and measured the force needed to do it.

The initial conditions that allow the procedure to begin are shown in the image below.

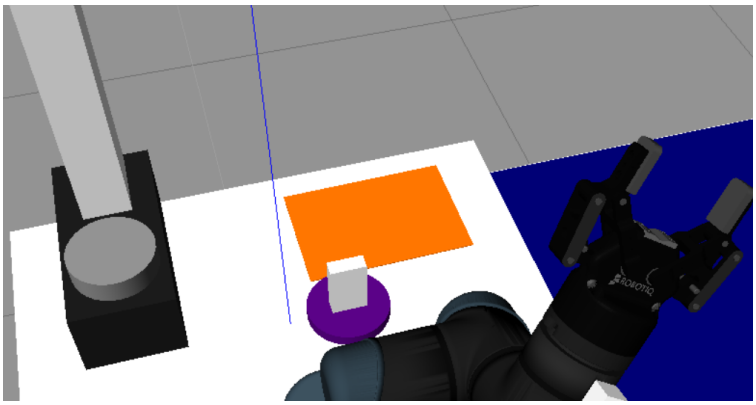

Figure 4.4. The figure shows the vacuum gripper in purple base and the plastic sheets in orange.

The principal steps of these operations are shown in the image below. You can see the first plastic sheet, than the pouch positioned above, and finally the placement of the second plastic sheet.

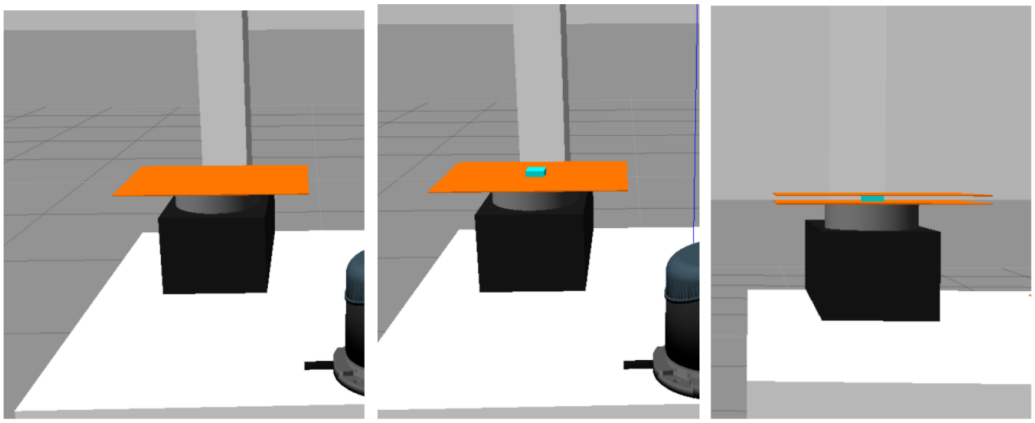

Figure 4.5. Strength procedure: principal steps using two plastic sheets and a pouch inside

Strength procedure

1. Place the vacuum gripper on the top of plastic sheets with the crushed pouch in the middle.
2. Activate vacuum gripper
3. Pick plastic sheets with the crushed pouch in the middle from Mark10 Tigthness.
4. Place everything in the bin.
5. De-activate vacuum gripper.
6. Put back the vacuum gripper on its holder.

The final result of these operations is shown in the image below.

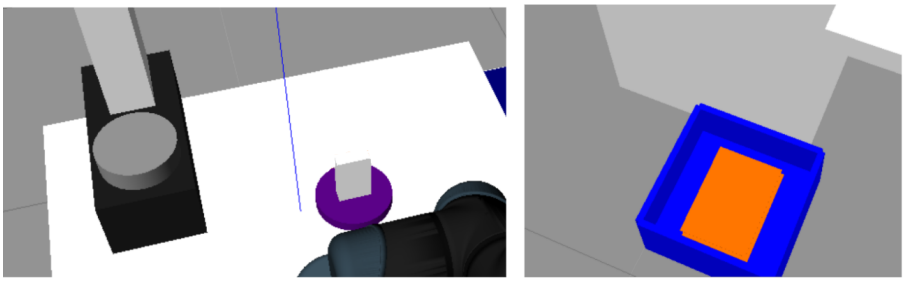

Figure 4.6. The figure shows the result at the end of the whole procedure.The vacuum gripper is on its holder and the plastic sheets with the crushed pouch are in bin in blue.

Note:
this plan can be generated by an automatic planner, using for example the Universal Planning Framework (UPF) developed within the AIPlan4EU project.

4.2 Analysis of the failures

The one presented in the previous chapter is the "ideal" nominal plan, that assume that all is going well, and there are no inconveniences or unforeseen events that interrupt the succession of the events foreseen, and bring the plan to failure. In the real world this condition is not always true, sometimes accidents that leading to the failure of the plan can occur: such as the pouch falling from the gripper during a move (look at the figure below 4.7); and it is useful and necessary to find a way dealt with it, in one way or another.

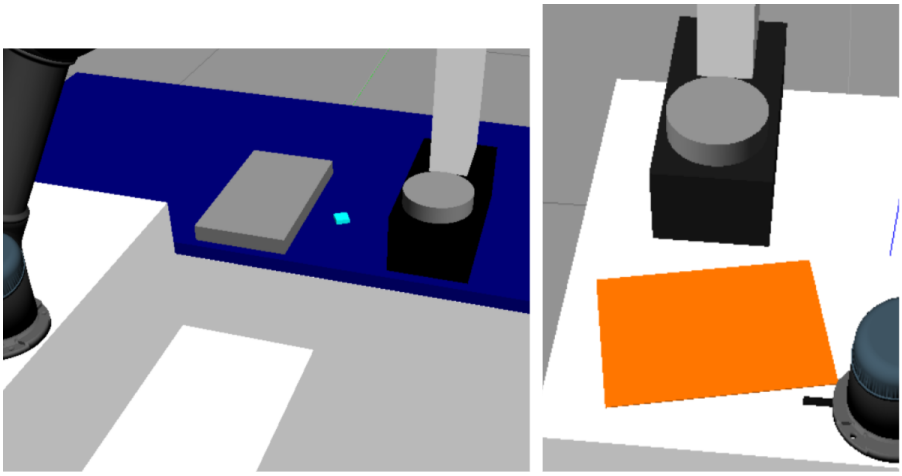

Figure 4.7. The figure shows the fall of the pouch when gripper moving from one tool to another, and, the fall of the plastic sheet from vacuum gripper

I found it useful to divide and study the possible cases of failure of the nominal plan into 4 macro areas:

1. Handling failure:

   during the picking and transporting operations the pouch, or the plastic sheet, falls down from the gripper.

2. Contact failure :

   during plan procedures, the robot hits something external; e.g. human or instrument.

3. Internal system failure :

   the robot has a breakdown related to internal system; e.g.: engine overload, stuck joints, sensor fault detection, software problems, ...

4. External system failure :

   the instrumentation (scale, Mark10 Strength, Mark10 Tigthness, or vacuum gripper) does not work correctly; or the pouch (or plastic sheet) have run out and the tray is empty.

Working on a simulator; in this thesis I focused on analyzing and finding a recovery solution procedure for the first two areas: handling and contact failure, and the condition: <pouch have run out and the tray is empty>. For the rest, I assumed that internal (engine, joints, sensors, ...) and external (scale, ...) instrumentation always worked correctly.

4.3 Proposed solution

My proposed solution to the failure problems is developed using procedures that include the presence of human-in-the-loop. More in detail, if during the execution of the nominal plan an accident occurs and the robot notices the problem[1], at first it will carry out a series of actions to try to recover and resume by itself the nominal plan; if this fails, robot will send a signal for external human intervention. At this point, generally, the human will find himself in front of three possibilities:

- send a series of commands or execute physical actions that allow robot to recover and continue the nominal plan;
- stop everything and make the plan fail (plan terminates with failure);
- restart the plan all over.

Note: I assumed that the human always takes the "right action", that his intervention does not make mistakes that prevent the recovery and the correct continuation of the plan and its goals.

How this solution has been implemented is shown in the next chapter; focusing the works on the Weighing procedure, defining a series of techniques and codes that can easily be extended to the others.

The implementation process will start showing the case of the "optimistic" nominal plan of Weighing procedure, for which checks and recoveries are not necessary because everything works as expected. To then get to build a more complex nominal plans with sophisticated control conditions and recovery solutions.

[1]Note : what I tried to pay attention to is avoiding the situation that the robot does not notice a problem and continues the nominal plan, assuming that the sensors work perfectly. The case in which some sensors fail, can only be detected by a periodic check of the operations by a qualified technician; for example every 45 min he verifies that the plan is proceeding as established.

Chapter 5

AI Planning Implementation and Results

5.1 Implementation design and aim

As anticipated in the previous chapter, i will begin to implement the Weighing procedure (Note: techniques and codes exposed here can easily be extended to the others procedure of nominal plan) in its version "optimistic": in which checks are not necessary because everything works as expected.

What I did was determine 6 way-points in space, which the center of the gripper must reach, clearly giving appropriate values to the 6 joints of the UR5 :

- in three of these way-points an elaboration takes place: vision process (to extract the exact position of the pouch in 3D coordinates) in point called `vision`; gripper closing and opening respectively in point called `pickpouch` and `scale`;
- two serve as checkpoints (their utility will be clearer when i expose the recovery plan for manipulation), called `med1` and `med2`:
- the last one simply allows not to interfere with the weighing measure, moving the gripper away from the scale; it is called `leave`.

The path of the gripper center implemented is schematized in the graph below, in which it possible to see the progression of its z coordinate over time.

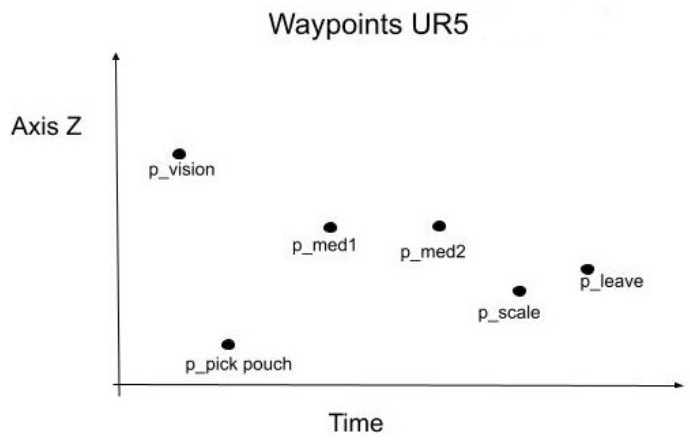

Figure 5.1. The graphic shows the path of the gripper center in the Weighing plan implemented according to the z axis (in practice the height of the space) over time.

The final result of this implementation phase will be: starting from the vision processing position on the tray, pick the pouch, move it through the checkpoints and place on the scale; then there will be a few seconds to wait to allow for the conclusion of the weighing process.

5.2 ROS Action

I have implemented all code of the plan using the library `actionlib`[1], for this reason I want to dedicate a brief description about it, to allow the reader, who encounters the concept of ROS actions for the first time, to proceed in reading the codes of the thesis.

ROS actions are a type of ROS implementation for achieving goal-oriented behavior; used with large and complex systems, these turns out very flexible and effective, especially when the classic ROS service takes a long time to execute, and the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.

ROS actions follow a client-server paradigm: `actionlib` provides the tools and an interface for interaction between two nodes: the action client node and the action server node. The client has the task of sending control signals (will send a goal, or a list of goals, or cancel all the goals), whereas the server has the task of listening to those control signals and providing the feedback (would return the status of that goal, the result upon the completion of a goal, and feedback: that is periodic information about the goal being achieved or cancelled). In order for the client and server to communicate, it is necessary to define a few messages on which they communicate: Goal, Feedback, and Result messages, each of them is specified by a type of ROS message. These are stored in a file with the extension `.action` [2].

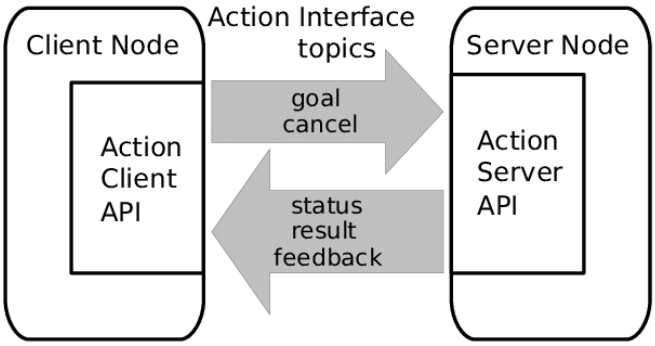

Figure 5.2. The diagram gives a general and schematic idea of client-server concept behind ROS Actions [27]

To address this, i will show the structures and functions necessary for the movement of the robotic arm UR5, used in the realization of the codes of this thesis work.

[1] http://wiki.ros.org/actionlib

[2] Note: from this file, several messages are generated and used internally by actionlib to communicate between the action client and the action server

Starting from Rostopic list of the simulator that I built in this thesis:

```
/trajectory_controller/command
/trajectory_controller/follow_joint_trajectory/cancel
/trajectory_controller/follow_joint_trajectory/feedback
/trajectory_controller/follow_joint_trajectory/goal
/trajectory_controller/follow_joint_trajectory/result
/trajectory_controller/follow_joint_trajectory/status
/trajectory_controller/state
```

Figure 5.3. Part of the Rostopic list of simulator.

The topics that end with: `/goal` `/cancel` `/result` `/status` `/feedback` are ROS action implementation.

`/joint_trajectory_controller` is used for executing joint space trajectories on a list of given joints. The trajectories to the controller are sent by means of the control messages: `FollowJointTrajectoryAction` action interface in the `follow_joint_trajectory` namespace of the controller.

The `FollowJointTrajectory` action server is already implemented as part of the Gazebo node. The topics ending with `/feedback` `/result` `/status` are published by the Gazebo node (alias, the robot) and the topics ending with `/cancel` `/goal` are being subscribed to by the robot. Practically, to move the robot arm, will be send a goal to the action server. To better understand how it all works, I will show and explain the essential part of Python code used for this purpose, via an action client implementation.

Let's start with the imported libraries :

```
import actionlib

from control_msgs.msg import FollowJointTrajectoryAction,
  FollowJointTrajectoryGoal,JointTrajectoryAction,
  JointTrajectoryGoal

from trajectory_msgs.msg import JointTrajectoryPoint,
  JointTrajectory
```

The first is introduced for get the SimpleActionClient; the next ones are needed to import messages used by action: for send goals, receive feedback, ect ... .

Note, at the beginning of the code is necessary initializes a rospy node so that the SimpleActionClient can publish and subscribe over ROS:

```
  rospy.init_node("arm_moving", anonymous=True)
```

Then is necessary to initialize the action client passing the name and the type of the action.

```
 name = 'trajectory_controller/follow_joint_trajectory'

 client = actionlib.SimpleActionClient(name,
      FollowJointTrajectoryAction)
```

Note: the action client and server communicate over a set of topics. The action name describes the namespace containing these topics, and the action specification message describes what messages should be passed along these topics.

Then is necessary to wait for a response from the action server: only once action server is activated, it is ready to be receptive to goals.

```
client.wait_for_server()
```

Now it is possible to define a goal to send to the action server.

```
joint_pose = [-0.1, -2.0, 2.2,-1.75,-1.57,1.57]

goal = FollowJointTrajectoryGoal()

goal.trajectory = JointTrajectory()

goal.trajectory.joint_names = ['shoulder_pan_joint',
'shoulder_lift_joint', 'elbow_joint',
'wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']

goal.trajectory.points =
[JointTrajectoryPoint(positions=joint_pose,
velocities=[0]*6,
time_from_start=rospy.Duration(4.0))]
```

It is necessary to pass information via the structure `FollowJointTrajectoryGoal()` message, which needs the name of single joints, the value of angle of single of joints, velocity and the time (4 second in in the above case) to move the arm to a specified trajectory.

Once the goal is defined, the command below sends this to the action server.

```
client.send_goal(goal)
```

While with command below, allow to wait for the server to finish performing the action.

```
client.wait_for_result()
```

For prints out the result of executing the action.

```
client.get_result()
```

* * *

The same concepts and principles were used to close (sending value 0.00) and open (value 0.85) the simulator gripper; very briefly:

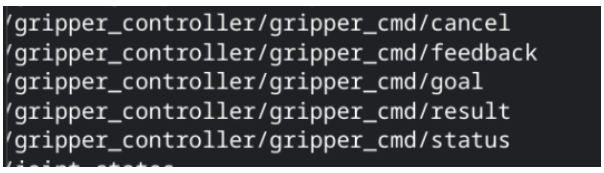

Figure 5.4. Part of the Rostopic list of simulator.

and defined the following function:

```
import actionlib
import control_msgs.msg

def gripper_client(value):

      client = actionlib.SimpleActionClient(
          '/gripper_controller/gripper_cmd',
```

```
            control_msgs.msg.GripperCommandAction)

        client.wait_for_server()

        goal = control_msgs.msg.GripperCommandGoal()
        goal.command.position = 0.85 - value
        goal.command.max_effort = -5.0  # effort limit

        client.send_goal(goal)
        client.wait_for_result()

        return client.get_result()
```

Note: all the functions, for the movement of the robot arm or the gripper, that will be encountered later in this chapter, are based on this type of implementation just outlined.

5.3 A classical approach

A typical approach to the implementation of a simple plan like the one proposed above, involves the definition of a ROS node, which exploits some ROS library and uses multiple functions for individual tasks; basically builds a sequence of codes like this one:

```
Code initialization

move_arm(positions=[vision])
time.sleep (1.0)

pickpouch = extractLocation3Dpouch()

move_arm(positions=[pickpouch])
time.sleep(1.0)

gripper_client(value_close)
time.sleep(1.0)

move_arm(positions=[med1])
move_arm(positions=[med2])
move_arm(positions=[scale])
time.sleep(1.0)

gripper_client(value_open)
time.sleep(1.0)

move_arm(positions=[leave])
time.sleep(10.0)

Code closing
```

A code like this, works as expected, it has been tested on the simulator; it can be made more elegant, but it is quite understandable and substantially does its duty.

The first critical issues already emerge when the first conditions are introduced in flow control structure; see for instance :

```
Code initialization

While (flag_vision==True) and (not flag_collision==True)
and ... and (not flag_plan_finished==True):

    move_arm(positions=[point])

    if (point ==[vision]):
         pickpouch = extractLocation3Dpouch ()

    if (point ==[pickpouch]):
         gripper_client(value_close)
         time.sleep(1.0)
    if (point ==[scale]):
         gripper_client(value_open)
         time.sleep(1.0)
    ...
```

```
    point = next_point

    Update flags
    if flag1 == False:
        recovery_function1()
    if flag2 == False:
        recovery_function2()
    ...
    else:
        print("plan_is_fail")
        exit

Code closing
```

A code of this kind, even if effective, begins to become difficult to manage, not easy to understand by users other than the creator, difficult to adapt to the arrival of new needs or changes; put it simply this means: increase the maintenance costs of a program.

A solution to this need is presented in the next sessions.

5.4 Architecture of the implementation system

In this section is presented the software infrastructure which allows the interaction between the various components (alias container) used in the implementation phase of the plans:

- Xserver : for a graphic output of the simulator on the browser [3]
- Simulator : substantially the whole ROS (packages and libraries) and Gazebo system described in detail the section 3.2 and 3.3
- Petri Net Plans container: for execute the plan (it is the Plan Execution Monitor). More in detail: PNP publishes on a String topic : `/pnp/actions_str`, a message that triggers a thread initialization for each action. Action executors listens this message and execute the corresponding action. In more, PNP reads conditions (alias fluents) from ROS parameters with values: 1 (True), 0 (False), -1 (Unknown). Note: condition executors set these values.

All system is managed and launched according to Docker containerization paradigm, more precisely, through the use of Docker Compose[4]. This tool allows to define and share multi-container applications in an extremely simple and quick way; just by defining a `.yml` file, more services can be launched and controlled with a single command. See the Appendix A.8 to have an idea.

This type of architecture not only guarantees modularity to the entire system, a quick and easy-to-understand configuration, but also the possibility of using the individual blocks as if they were "black boxes" (i.e.: it is not necessary to know in detail its internal dynamics, this may be unknown, it affects only inputs and outputs).

Figure 5.5. The figure schematizes the architecture of the implementation system. The diagram summarizes the relationships between the containers.

[3] Exactly in: http://localhost:3000

[4] https://docs.docker.com/compose/

5.5 Test case 1

To guarantee modularity, high expressiveness, and flexibility in case of changes; the plan Weighing procedure, its version "optimistic", was formalized using Petri Net Plans (for more detail see Section 3.7) and Plan Execution Interface (Section 3.8) technology. The result is shown below and stored in file with extension `.plan` .

Note: due to its "optimistic" nature, the plan contains only actions; there aren't fluents either conditions that check or change their status over time.

```
goto_vision;
goto_pickpouch;
movegripper_close;
wait_1;        #time consolidation gripper
goto_med1;
goto_med2;
goto_scale;
movegripper_open;
goto_leave;
wait_10        #time to measure and record weight
```

The first thing you notice is the compactness and immediacy of the text. The changes are easy and immediate, just add a line. Understanding also becomes easier compared to a typical implementation.

Note: `vision` is location that allows the wrist camera to frame the entire tray; at this point an action of the type: `extractLocation3Dpouch` should be activated which allows obtaining the 3D coordinates of the pouch and setting the value of parameter: `pickpouch`. For the purposes of this first test, it was preferred to give a priori (i.e., it does not require processing, just given in input by the programmer) to this parameter the value of initial position of pouch in tray.

This plan was launched in the simulator with a successful result. See the frame sequence below:

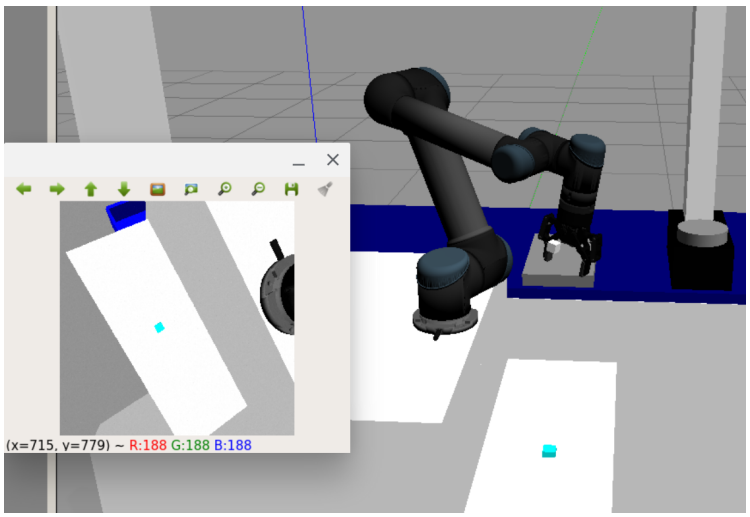

Figure 5.6. The figure shows the start up step of Weighing procedure: place the robot wrist camera on top of the tray so that its contents are fully visible (point vision)

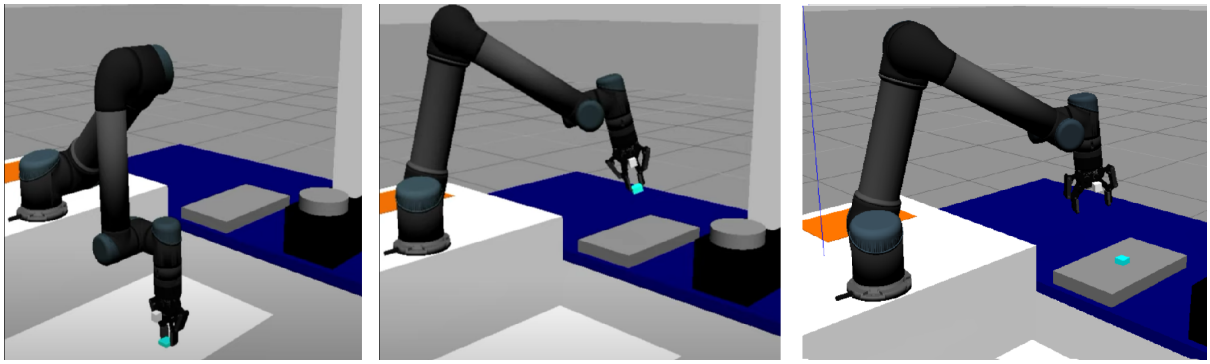

Figure 5.7. The figure shows three steps of Weighing procedure: the moment in which the pouch is taken (point pickpouch), the moment of transport to the second check point (point med2), when gripper moves away from the scale and waits for the weighing (point leave)

5.6 Test case 2

In this section, always taking advantage of the technology PNP and PLEXI learned, a more complex plan will be presented; which, in addition to the actions, makes use of control conditions, fluents, human-machine interaction and recovery plans. More in detail, in addition to that seen in Test case 1:

- there will be an initial control to verify the presence of the pouch in the tray: `sense_presentpouch`; if pouch is missing, the intervention of the human will be expected to fill the tray and then proceed with the plan. In this simulation case, the ROS node described in the section A.5 will be used to refill the tray.
- Then a further check, formalized in an execution rules in the file of plan (to be applied only a specific action):

  ```
  *if* (not grasppouch) *do* recoveryManipulation; restart_action
  ```

  This is to verify that the pouch is grabbed by gripper during the move. In the event of a fall, the recoveryManipulation plan is evoked. In this simulation case, fall event was simulated by creating a ROS node, with a timer, that slightly opens the gripper when passing between points med1 and med2. If the recoveryManipulation plan is successful, the interrupted action restarts.
- There will also be an execution rule separate from the file plan, with the extension `.er`; in this case the execution rule are applied to each action `goto` of the plan, which will verify a collision occurred. At the moment, this is purely theoretical: as it requires the plugin of a laser scanner simulator that goes beyond the time limits imposed by this thesis; but the thing itself is feasible in a future works.

Implementation note: not having vision elaborations at my disposal, as training made on the images of the simulator's camera are considered unusable in the transition to the real world; it was decided to create a node, described in the

section A.6 that uses a detection structure of the "ground truth" of Gazebo space. All the part that on the real robot requires an elaboration of vision; here, on the simulator was processed with this method; both for the check and set of parameters in: `presentpouch`, `extractLocation3Dpouch`, `grasppouch`. In future works in real scenarios, the structure of these clearly will be redefined.

The plan file:

```
goto_vision;

sense_presentpouch;
< presentpouch? say_PouchPresent :
  (not presentpouch)? say_HelpRequestMissPouch;
   waitfor_proceedSignal
>;

extractLocation3Dpouch;    # set value of _pickpouch
goto_pickpouch;
movegripper_close;
wait_1;

goto_med1;  ! *if* (not grasppouch) *do* recoveryManipulation;
              restart_action !
goto_med2;  ! *if* (not grasppouch) *do* recoveryManipulation;
              restart_action !
goto_scale; ! *if* (not grasppouch) *do* recoveryManipulation;
              restart_action !

movegripper_open;
goto_leave;
wait_10
```

In the phase `movegripper_close` if gripper does not grasp the pouch it's notice in the `med1` phase, so no further checks are required in this line.

The action `waitfor_proceedSignal`, allow to wait for an input signal from the user to continue. It is basically a Python input function[5].

The action `say_`, at moment, is a simple screen print, but in the future it can become speech; as well as the entire human-machine interaction.

The execution rule file:

```
# Execution Rules for ER domain
# for nominal plan

*if* collision *during* goto *do* say_NotifiedCollision;
waitfor_proceedSignal; sense_AskRemovableObstacle;
< AskRemovableObstacle? say_WaitingForFreeSpace;
    waitfor_proceedSignal; restart_action :
  (not AskRemovableObstacle)? ModulLbD;
    say_NotifiedNewTrajectory; waitfor_proceedSignal;
  restart_plan >
```

As previously said, this ER can take place only in the presence of a collision detector.

Practically what happens here, in the event that a collision is detected, there are two possibilities: the impacted object can be moved away, the human intervenes

[5] raw_input("Press Enter to continue...")

and resolves the situation, and the path can continue. If the object is immovable, through the processing of a Learning by Demonstration module, new values are given to the various parameters of action `goto`: `med1`,..., `scale`, ... (so recalculated the trajectory), to then restart the plan.

Note: the fleunt `AskRemovableObstacle` is True, if user types `yes` in keyboard input function during `sense` action.

As regards the recoveryManipulation plan, my starting idea was to create a spiral trajectory, traveled by the gripper, to look for the pouch: starting from the point where the action is interrupted (detecting this position as the center of the gripper), getting some way-points obtained from a spiral function centered on the break-point, move the arm UR5 for reach these points and, with a vision processing, try to identify the pouch and pick it. Something like that:

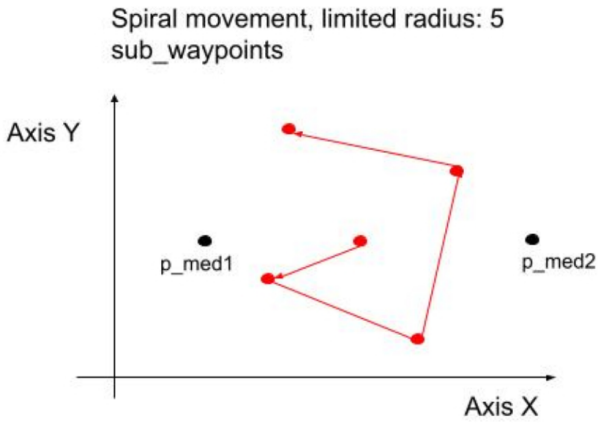

Figure 5.8. The figure shows the spiral trajectory proposed for recoveryManipulation plan. As an example, the pouch is considered to have fallen while the robotic arm was moving towards the med2 point.

The radius of the spiral should be statistically defined on an experimental basis.

This solution, in a simulator that doesn't do vision processing, can be simplified to just two points. So, the file of recoveryManipulation plan has this form:

```
movegripper_open;
goto_spiral1;
goto_spiral2;
say_helpRequestMissPouch;
sense_AskRestarPlan;
< AskRestarPlan?
    waitfor_proceedSignal;
    restart_plan :
  (not  AskRestarPlan )?
    fail_plan
>
LABEL_RMW
```

In a real experiment on the robot, the proposed plan can easily be extended by adding action lines: `goto_spiral3`, `goto_spiral4`, ...

And its associated ER file:

```
# Execution Rules for ER domain
# for recoveryManipulation

*if* presentpouch *during* goto *do* extractLocation3Dpouch;
   goto_pickpouchRecovery; movegripper_close; sense_grasppouch;
 < grasppouch ? GOTO_LABEL_RMW :
  (not grasppouch) say_NotifiedProblemExtraction;
    waitfor_proceedSignal; fail_plan >
```

In practice, during the `goto_` action, the presence of the pouch is checked, if present it is verified, it try to grab it, if it succeeds (it is confirmed by `grasppouch`), the recovery plan is exited from the `LABEL_RMW` and the action of the nominal plan restarts. Otherwise, the recoveryManipulation plan comes to an end by notifying the loss, and asking the user what to do: if he wants to restart the plan or make it fail.

Note: The ER, referred to the all action of `goto` in nominal plan (for the detection of collisions), can be simplified in its logic structure and added in file of ER of recoveryManipulation, in this way:

```
# Execution Rules for ER domain
# for recoveryManipulation

*if* presentpouch *during* goto *do* extractLocation3Dpouch;
   goto_pickpouchRecovery; movegripper_close; sense_grasppouch;
 < grasppouch ? GOTO_LABEL_RMW :
  (not grasppouch) say_NotifiedProblemExtraction;
  waitfor_proceedSignal; fail_plan >


*if* collision *during* goto *do* say_NotifiedCollision;
   waitfor_proceedSignal; sense_AskRemovableObstacle;
< AskRemovableObstacle? say_WaitingForFreeSpace;
waitfor_proceedSignal; restart_action :
  (not AskRemovableObstacle)? fail_plan >
```

The images on the following pages give an idea of the one just exposed.

The Case one shows the situation in which the nominal plan has a break, the pouch fell off the gripper, and recoveryManipulation plan applied has been successful: the nominal plan was taken up autonomously by the robot.

The Case two, instead, show when recoveryManipulation plan fails because pouch has been lost, and human intervention is required.

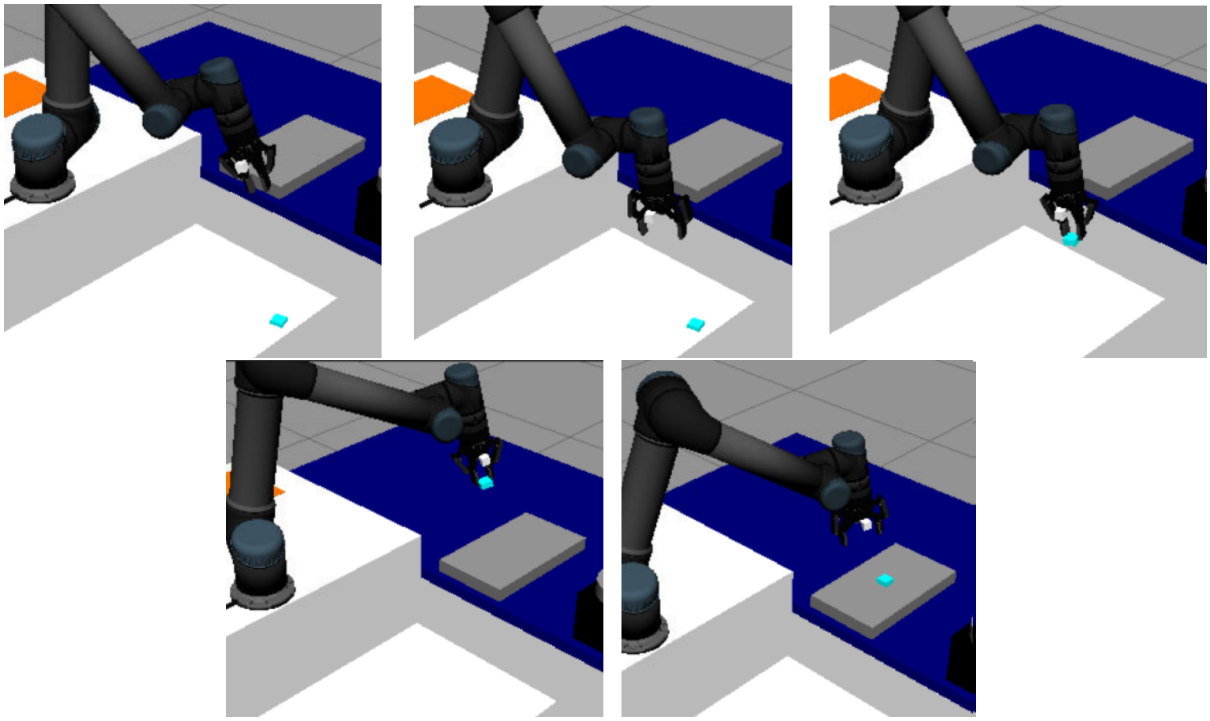

Figure 5.9. Case one : Frame1, gripper lost the pouch as it moves towards the point med2. Frame2, start recoveryManipulation plan. Frame3, the pouch has been found and is picked. Frame4 and Frame5 the nominal plan resumes and comes to the end.

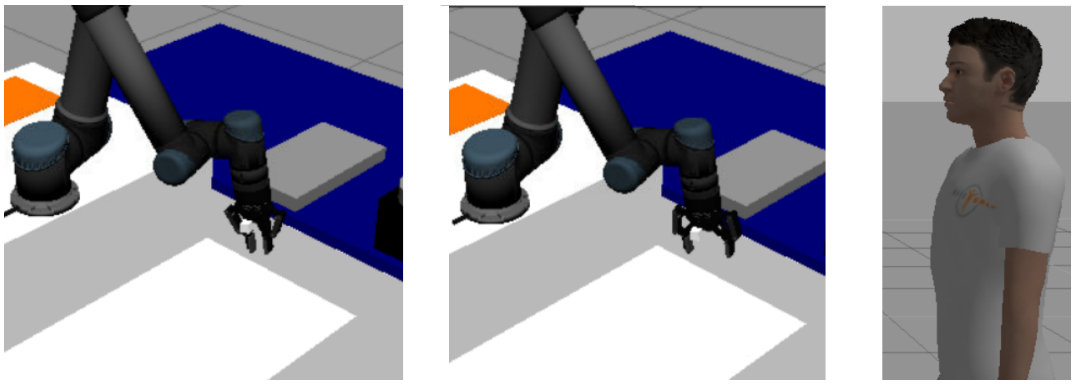

Figure 5.10. Case two : pouch has been lost, it has fallen to the ground and is out of reach for the robotic arm UR5. Human intervention is required (Human-in-the-loop).

5.7 Test case 3

In this session I will propose a solution that allows a more dynamic and intuitive human-robot interaction than the screen printing shown in the previous test case; without complicating the implementation aspect of the codes.

For this purpose Multi-mODal Interaction Manager (MODIM)[6] technology will be used. Always taking advantage of the Docker Compose paradigm exposed in the session 5.4, MODIM application integrates[7] into the architecture of the implementation system as a container that allows both: the interaction with the user via browser and dynamic changes of the plan. By way of example, to explain the MODIM logic and its advantages in this context, I will consider the case of recoveryManipulation plan presented the previous session. Here, this recovery plan is rethought using the MODIM framework, obtaining the same results of the previous formulation, but a easier interaction with the user, and a greater compactness of the code:

```
modim_init;
modim_informRM;

movegripper_open;
goto_spiral 1;
goto_spiral 2;

modim_ask_question;
< modim_yes? restart_plan :
  modim_no? fail_plan :
>;

LABEL_RMW
```

Following some comments to explain the code:

`modim_init` is a function used to start the state of the GUI. It requires a configuration file called `init`: a plain text file, in which the user is initialized; in this case is:

```
URL: index.html
PROFILE: <*,*,en,*>
TEXT: WELCOME
IMAGE: img/someimage.jpg
```

`informRM` is an example of MODIM action, this inform of the start of the recovery plan, displaying the information as text. It has a very simple structure:

```
IMAGE
<*,*,*,*>:  img/recoveryM.jpg
----
TEXT
<*,*,*,*>:  Recovery Manipulation plan it started
----
```

Where possible, to get speech, simply by adding the line at the end (TTS: Text-to-speech) :

[6] https://bitbucket.org/mtlazaro/modim/src/master/

[7] Which takes the form of some command line in the yml file

```
TTS
<*,*,*,*>:   Recovery Manipulation plan it started
----
```

The interactive part begins with `modim_ask_question`. If the recovery plan get at this point, it means that the pouch can not be recovered autonomously by the robot, and the intervention of human is required. Something like this should appear on the screen (via browser):

Figure 5.11. Output display of MODIM action: question

A user response is waited by clicking on Yes for restart the plan, or, No for fail the plan and quit.

The code structure of `question` is similar to the previous one, with the addition of the BUTTONS functionality (to set of possible answers from the user):

```
IMAGE
<*,*,*,*>:  img/recoveyM2jpg
----
TEXT
<*,*,*,*>:  Do you want restart the plan?
----
BUTTONS
yes
<*,*,*,*>:  Yes
no
<*,*,*,*>:  No
```

With MODIM is also possible to take the advantages offered by the logic of execution rules: an emergency button “Stop” can be added on screen, with which the user can interrupt the execution of the plan during the movements of the robotic arm.

```
*if* modim_stop *during* goto *do* modim_init; fail_plan
```

5.8 Future works

In this last session, I suggest some directions and objectives to work on in the near future:

- Experimenting with the AI plans formulated in this thesis on the real robot; in real lab scenario; developing adequate vision processes (using the real camera placed on the robot's wrist) and exploiting the functionality of the collision detector (also present on the real robot), as proposed in the session 5.6.
- Optimize the nominal and recovery plans proposed with experimentation on real robots through statistical analysis.
- Addressing other procedures of nominal plan: Tightness procedure, Strength procedure, Cleaning procedure; refer to the section 4.1.
- Design and test recovery plans that also include the cases : external system failure and internal system failure; refer to the section 4.2.
- Develop modules of Learning from Demonstration, which would allow to extend and adapt more easily the capabilities of the robot to new situations.
- Develop a more interactive human-robot interface, which makes the interaction more friendly; for example, whole interaction can take place through speech process.

Chapter 6

Conclusions

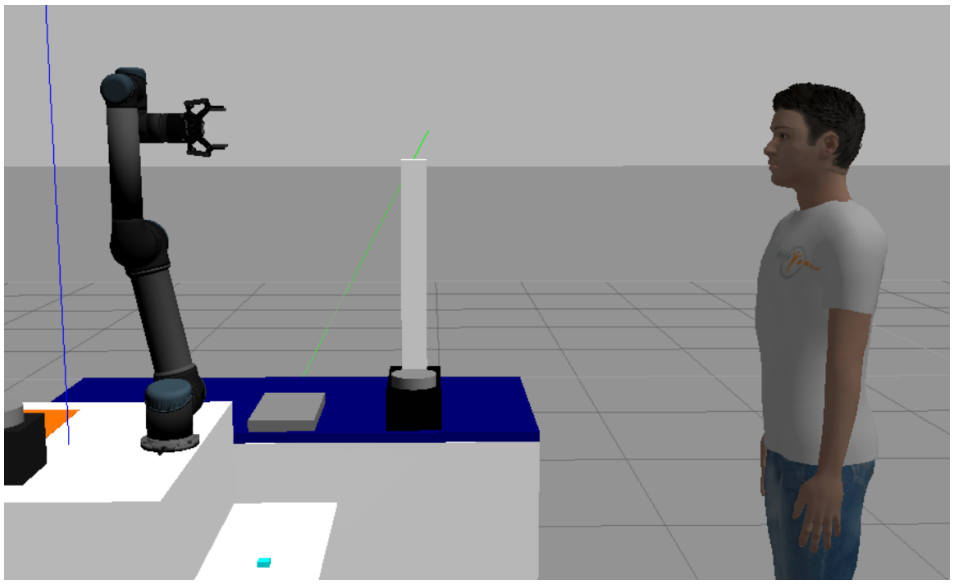

Figure 6.1. Final ornamental image in a perspective AI planning with Human-in-the-loop.

In this master's thesis, I presented the results of my work in the realization of a robot simulator (using development tools such as ROS and Gazebo, within of Docker containerization technology) that, by reproducing all the fundamental aspects of the Procter & Gamble real laboratory, allowed me to design, implement and test a series of AI plans for industrial quality control test. AI plans are formalized with a technology (Petri Net Plans and PlanExecution Interface) that allows to guarantee modularity, high expressiveness,and flexibility in case of changes; and of course to satisfy the purposes proposed by Procter & Gamble.

Taking into account, unforeseen events and failures, recovery solutions were proposed for the nominal plan. These were centered around the concept of Human-in-the-loop.

This page blank left intentionally

Appendix A

A.1 How to run GUI applications in a docker container

Below is shown the Linux configuration used to launch a docker thesis's container which allow to run Gazebo graphical simulation inside:

```
docker run -it \
    --net=host \
    --env="DISPLAY" \
    --env="QT_X11_NO_MITSHM=1" \
    --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    --volume="$HOME/.Xauthority:/root/.Xauthority:rw" \
    --device=/dev/dri:/dev/dri \
    --privileged \
    ID_IMAGE bash
```

Some comments:

```
--net=host
```

Container shares networking with the host OS.

```
--env="DISPLAY"
```

Provide the container with a DISPLAY environment variable (practically share the display of the host on the container)

```
--volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
```

X Window System, also known as X11 o simply X, is a program (for Unix-like systems) that manages graphics displays and input devices like keyboard, mouse, ... connected to the computer. It works as a server and can be run on the local computer or on another computer on the network. Services can communicate with the X server to display graphical interfaces and receive user input.
In this case, it is necessary to transfer the X11 socket from Linux local machine to the container in order to run directly.

```
--volume="$HOME/.Xauthority:/root/.Xauthority:rw"
```

Share the host's X server to the container OS (by creating a volume).

```
--device =/dev/dri:/dev/dri
```

Command for hardware transcoding

A.2 Specifications of the dockers container used for simulation

Main framework running inside the thesis's docker container are:

- OS : Ubuntu 16.04.5 LTS
- ROS kinetic 1.12.14
- Gazebo 7.

To allow the robotic simulator to run correctly with all its specifications and tools, a long and laborious analysis work of set-up was required. This led to remove old ROS repository GPG key and update it, to be able to update the packages of system:

```
apt-key del 421C365BD9FF1F717815A3895523BAEEB01FA116

apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80'
--recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

apt clean && apt update
```

The installation of an Gazebo upgrade:

```
apt-get install gazebo7
```

And addition of the installing the following packages for the following purposes :

For the proper functioning of the gripper:

```
apt-get install ros-kinetic-ros-control
ros-kinetic-ros-controller
```

For camera and other Gazebo plugins in ROS; more generally, packages provide the necessary interfaces to simulate a robot in Gazebo using ROS:

```
apt-get install ros-kinetic-gazebo-ros-pkgs
ros-kinetic-gazebo-ros-control

apt-get install ros-kinetic-image-view
```

For the construction of the simulation environment:

```
apt-get install git
git clone https://github.com/osrf/gazebo_models
```

In the end, installation of a text editor:

```
apt-get install nano
```

The docker thesis's image has been pushed to Docker Hub; this allow to store and share it with the community:

https://hub.docker.com/r/mariofiorino/thesis_ur5_gripper2f_lab_plan

A.3 Docker user's manual

This section contains the main commands (used in this thesis work) for managing the Docker system in Linux terminal. For Docker installation I suggest to follow the instructions of the site: https://docs.docker.com/get-docker/ and choose the distribution according to your operating system. I have used Linux - Debian 10 : https://docs.docker.com/engine/install/debian/

- To pull an existing project from Docker Hub registry on your machine, command line to write in the terminal is:

```
docker pull <ID-image>
```

- To see on screen the list of containers or images stored in your local memory:

```
docker container ls -a

docker image ls -a
```

- To remove specific container or image:

```
docker container rm <ID-container>

docker image rm  <ID-image>
```

- To remove all unused containers, images (all unused images not just dangling ones), volumes.

```
docker system prune -a
```

- To start a container on the list:

```
docker start -ai <ID-container>
```

  Note: after launched this line, you are in container specified. To run ROS - Gazebo system is necessary set up the environment (and to do this every time you start); in my thesis's container, once inside, write in the terminal:

```
cd ~/catkin_ws
source devel/setup.bash
```

  At this point the simulator can be launched:

```
roslaunch ur5_gazebo ur5_setup.launch
```

- To work in parallel on a running container using another terminal:

```
docker exec -it <ID-container> bash
```

  Note: in this case two "source" are required (the order is important):

```
cd /
source ros_entrypoint.sh

cd ~/catkin_ws
source devel/setup.bash
```

- To stop one or more running containers

```
docker stop <ID-container>
```

- To copy a file or folder from the container's system to the local machine:

```
docker cp <ID-container>:/file/path/within_container
/host/folder
```

  The other way around:

```
docker cp /file/local <ID-container>:
/dest/within_container
```

- To push the docker image from your machine on Docker Hub, allowing you share image with Docker community:

```
docker login -u "username-DockerHub" -p
"password-DockerHub-account" docker.io
docker push <ID-image>
docker logout
```

  Note: to be successful, you must first name your local image using your Docker Hub username; e.g. : my Docker Hub username is "mariofiorino", so the command line for push is something like this:

```
docker push mariofiorino/thesis_ur5_lab
```

- To display information of the amount of disk space used by the docker daemon:

```
docker system df
```

A.4 Simulation Description Format (SDF)

By way of example, in this section is shown the code that allows the simulation of the model the Mark10 Strength, with number (4) in Figure 3.5; in the file `ur5_setup.world` [1]. After reading the paragraph 3.3, the code should be easy to interpret. This is basically divided into three parts, each for the structures that make up the model: the base (`name="base"`), the tower (`name="towerS"`)and the cylinder (`name="cil"`). For each of these they are developed collision elements and visual elements. Below is a description of the main tags used:

- The tag `<pose>` defines the position x, y, z and the orientation according to the respective three axes, of the object within the frame world of Gazebo space.
- The tags `<box>` or `<cylinder>` determine the geometry of the object: parallelepiped or cylinder, and its measures : `<size>` x, y, z for parallelepiped; `<radius>` and `<length>` for cylinder.
- The tag `<friction>`: when two object collide, a friction term is generated. In ODE this is composed of two parts: `<mu>` is the Coulomb friction coefficient for the first friction direction, and `<mu2>` is the friction coefficient for the second friction direction (perpendicular to the first friction direction).
- The tag `<material >` controls the color components of an object.

```
<model name="MarkS10">
    <static>true</static>
    <link name="link">
      <collision name="base">
        <pose>-0.17 -0.252 1.0655 0 0 0</pose>
        <geometry>
          <box>
            <size>0.345 0.165 0.125</size>
          </box>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.6</mu>
              <mu2>0.6</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
      <visual name="base_V">
        <pose>-0.17 -0.252 1.0655 0 0 0</pose>
        <geometry>
          <box>
            <size>0.345 0.165 0.125</size>
          </box>
        </geometry>
```

[1] In path of thesis Docker container: `/root/catkin_ws/src/ur5/ur5_gazebo/worlds`

```
        <material>
          <script>
            <uri>file://media/materials/scripts/
            gazebo.material</uri>
            <name>Gazebo/FlatBlack</name>
          </script>
        </material>
      </visual>

      <collision name="towerS">
        <pose>-0.27 -0.252 1.465 0 0 0</pose>
        <geometry>
          <box>
            <size>0.13 0.08 0.675</size>
          </box>
        </geometry>
      </collision>
      <visual name="towerS_V">
        <pose>-0.27 -0.252 1.465 0 0 0</pose>
        <geometry>
          <box>
            <size>0.13 0.08 0.675</size>
          </box>
        </geometry>
        <material>
          <script>
            <uri>file://media/materials/scripts/
            gazebo.material</uri>
            <name>Gazebo/WhiteGlow</name>
          </script>
        </material>
      </visual>

      <collision name="cil">
        <pose>-0.09 -0.252 1.1465 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.07</radius>
            <length>0.04 </length>
          </cylinder>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.7</mu>
              <mu2>0.7</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
      <visual name="cil_V">
        <pose>-0.09 -0.252 1.1465 0 0 0</pose>
```

```
        <geometry>
          <cylinder>
            <radius>0.07</radius>
            <length>0.04</length>
          </cylinder>
        </geometry>
        <material>
          <script>
            <uri>file://media/materials/scripts/
            gazebo.material</uri>
            <name>Gazebo/Grey</name>
          </script>
        </material>
      </visual>
    </link>
  </model>
```

A.5 Adding models to Gazebo simulator

Pouches are soluble capsules contain liquid and powder, these are the center of the industrial quality control laboratory tests of Procter & Gamble Services Company. In Gazebo simulator they are represented by a small turquoise parallelepiped. During the simulation, it was necessary to add and move new pouches to specific places, without the execution of the simulation being interrupted and then restarted; to solve this problem I created a ROS node, which allows you to add a certain number of pouches (given as input by the user) in a specific position (also inserted input from the user) of the Gazebo space. This process is basically allowed by exploiting `gazebo_msgs/SpawnModel Service` [2]. Given in input the number of pouches to add (which determines the number of interaction of For-loop in which the core code is inserted) and the coordinates in Gazebo world to place the pouch, the core of node ROS is show below:

```
initial_pose = Pose()
initial_pose.position.x = x
initial_pose.position.y = y
initial_pose.position.z = z

stri = str(i+x+y)
model_name="pouch" + stri

rospy.wait_for_service('gazebo/spawn_sdf_model')

spawn_model_prox = rospy.ServiceProxy('gazebo/spawn_sdf_model',
             SpawnModel)

spawn_model_prox(model_name, sdff, "robotos_name_space",
            initial_pose, "world")
```

I believe that explaining the last line it explains the whole code, in particular the parameter given to `spawn_model_prox` :

- `model_name`, pass the name of model to Gazebo. This does not accept that models have the same name. So in order to avoid "Failure - model name already exist", I have devised a stratagem that allows to always give a new name to the model using the loop counter and the floating points coordinate x and y (generally always different from the previous one) : `stri = str(i+x+y)`
- `sdff`, points to the .sdf file of the model to spawn, in my case `pouch.sdf`.
- `"robotos_name_space"`, spawn robot and all ROS interfaces under this namespace; not much fundamental, can be also simply : `" "`.
- `initial_pose`, indicate the geometric coordinates for place the model in Gazebo space frame.
- `"world"`, allows to specify the Gazebo world.

To get an idea of the results obtained, two frames, taken at different times, are shown below. It is observed that it was possible to insert new pouches, in specific

[2] http://docs.ros.org/en/jade/api/gazebo_msgs/html/srv/SpawnModel.html

place, without interrupting and restarting the simulator running, but continuing to operate on it.

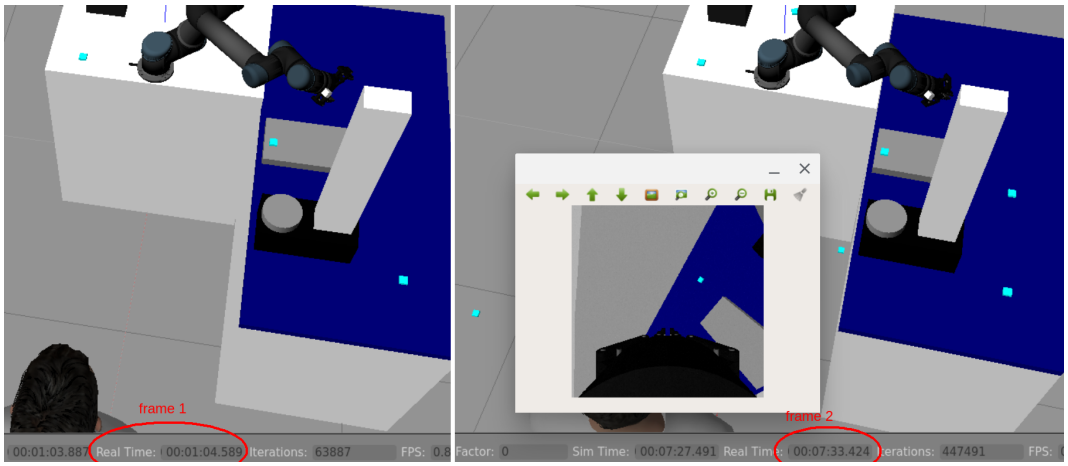

Figure A.1. Two frames taken at different times of the running Gazebo simulation. As stated above in Gazebo simulator pouch are represented by a small turquoise parallelepiped. The figure shows adding new pouches without interrupting the simulator running

A.6 ROS node : pouch detector

This ROS node allows to detect the position in the Gazebo space of any model given in input. In my case I needed to have the position of the "pouch" :

```
import rospy
import roslib
import rospkg
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import GetModelState


def main():

    rospy.init_node('get_pose')

    g_get_state = rospy.ServiceProxy("/gazebo/get_model_state",
    GetModelState)
    rospy.wait_for_service("/gazebo/get_model_state")

    pub = rospy.Publisher('/decect/pose', Pose)

    while not rospy.is_shutdown():
      try:
         state = g_get_state(model_name="pouch")

      except rospy.ServiceException:
        rospy.logerr(Error on calling service)
        return

      print(state.pose)
      pub.publish(state.pose)

if __name__ == '__main__':
  try:
    main()
  except rospy.ROSInterruptException: pass
```

Once launched with `rosrun`, depending on whether it is an action or a fluent, it is linked with a function of the type (from which to obtain the information requested on the pouch) :

```
    def __init__(self, ...):
        ...
        rospy.Subscriber('/decect/pose',Pose,self.callback)

    def callback(self,data):
         self.pose_x = data.position.x
         self.pose_y = data.position.y
         self.pose_z = data.position.z
```

For processing on fluent `grasppouch` the value `data.position` are compared with an established trajectory (to be precise, a range of values in which the expected trajectory is included).

A.7 goto_actionproxy file

As an example of action implementation, the file (in py) in its simplified form to facilitate understanding, that allows to move the robot arm, using the PLEXI formalism.

```
import rospy
import actionlib
from control_msgs.msg import ...
...
from  actionproxy  import  ActionProxy

ACTION_NAME = 'goto'

class GotoActionProxy(ActionProxy):

    def __init__(self, actionname):
        ActionProxy.__init__(self, actionname)
        self.movejoint = None
        self.movejoint = actionlib.SimpleActionClient(...)
        self.movejoint.wait_for_server()

        rospy.set_param('dgo', {
'vision'    : [-0.1, -2.0, 2.2,-1.75,-1.57,1.57],
'pickpouch' : [0.0, -0.75, 2.3,-3.14,-1.57,1.57],
'med1'      : [0.1, -1.3, 2.3,-3.14,-1.57,1.57 ],
... })

    def __del__(self):
        ActionProxy.__del__(self)

    def action_thread(self, params):
        joint_pose = rospy.get_param('dgo/%s'%params )

        goal = FollowJointTrajectoryGoal()
        goal.trajectory = JointTrajectory()
        goal.trajectory.joint_names = JOINT_NAMES
        goal.trajectory.points = [
           JointTrajectoryPoint(positions=joint_pose,
           velocities=[0]*6,
           time_from_start=rospy.Duration(4.0))]
        self.movejoint.send_goal(goal)

        finished = False
        while self.do_run and not finished:
            self.movejoint.wait_for_result()
            status = self.movejoint.get_state()
            result = self.movejoint.get_result()
            finished = (status == GoalStatus.SUCCEEDED)
                     or (status == GoalStatus.ABORTED)
```

```
if __name__ == "__main__":

    params = None
    if (len(sys.argv)>1):
        params = sys.argv[1]

    a = GotoActionProxy(ACTION_NAME)

    if params is not None:
        a.execute(params)
    else:
        a.run_server()
```

A.8 Docker compose file: yml

In this session, an simplified example of `.yml` file that allows the configuration of the entire system among its different components: Xserver, Simulator, Petri Net Plans container.

```
version: "2.3"
services:

  xserver:
    image: devrt/xserver
    container_name: xserver
    ipc: host
    security_opt:
      - seccomp:unconfined
    environment:
      - DISPLAY=:1
    ports:
      - "3000:80"

  simulator:
    image: mariofiorino/thesis_ur5_gripper2f_lab_plan
    container_name: simulator
    tty: true
    ipc: host
    network_mode: host
    security_opt:
      - seccomp:unconfined
    environment:
      - DISPLAY=:1
    volumes_from:
      - xserver
    depends_on:
      - xserver
    volumes:
      - $PWD:/root/
    working_dir: /root/
    entrypoint:
      - bash
      - -ci
      - sleep 10 &&
      roslaunch ur5_gazebo ur5_setup.launch paused:=true

  pnp:
    image: iocchi/pnp:kinetic
    container_name: pnp
    network_mode: host
    tty: true
    volumes:
      - $PWD:/home/robot/pg_lab
    entrypoint:
      - bash
```

```
      - -ci
      - sleep 15 && roscd pnp_ros && ./init.bash
```

Bibliography

[1] A. Oldershaw, L. Iocchi, M. Leonetti; A Case Study in Learning Plan Recovery Procedures from User Demonstrations, workshop of the 17th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2018), November 2018

[2] Del Duchetto, F., Kucukyilmaz, A., Iocchi, L., Hanheide, M.; Do not make the same mistakes again and again: Learning local recovery policies for navigation from human demonstrations. IEEE Robotics and Automation Letters 3(4), 4084–4091 (Oct 2018)

[3] Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal; Robot programming by demonstration. In Springer Handbook of Robotics of Bruno Siciliano, Oussama Khatib, Springer, 2008.

[4] Ravichandar, Polydoros, Chernova, Billard; Recent Advances in Robot Learning from Demonstration. Annual Review of Control, Robotics, and Autonomous Systems. 3. 10.1146/annurev-control-100819-063206; May 2020

[5] Iocchi, L., Jeanpierre, L., Lazaro, M.T., Mouaddib, A.; A practical framework for robust decision-theoretic planning and execution for service robots. In: Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016. pp. 486–494 (2016)

[6] Leonetti, M., Iocchi, L., Ramamoorthy, S.; Learning finite state controllers from simulation. In: European Workshop on Reinforcement Learning (EWRL) (2011)

[7] T. Murata. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4):541–580, 1989

[8] Ziparo, V.A., Iocchi, L., Lima, P.U., Nardi, D., Palamara, P.F.; Petri net plans - A framework for collaboration and coordination in multi-robot systems. Autonomous Agents and Multi-Agent Systems 23(3), 344–383 (2011)

[9] http://www.scholarpedia.org/article/Petri_net

[10] Dechsupa, Vatanawood, Thongtak. Transformation of the BPMN Design Model into a Colored Petri Net Using the Partitioning Approach. 2018

[11] https://github.com/iocchi/PetriNetPlans/wiki/Plan-syntax-for-PNP-generation

[12] C. E. Rasmussen and C. K. I. Williams; Gaussian Processes for Machine Learning, ser. Adaptative computation and machine learning series. The MIT Press, 2006.

[13] Andrea Lockerd and Cynthia Breazeal; Tutelage and socially guided robot learning. In Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, volume 4, pages 3475–3480. IEEE, 2004.

[14] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning; A survey of robot learning from demonstration. Robot. Auton. Syst.; May 2009

[15] Allen B. Tucker, Computer Science Handbook, second edition, Chapman & Hall/CRC 2004

[16] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Etivier, Sylvie Pesty. A Review of Learning Planning Action Models. Knowledge Engineering Review, Cambridge University Press (CUP), 2018.

[17] Ghallab, Malik; Nau, Dana S.; Traverso, Paolo, Automated Planning: Theory and Practice, Morgan Kaufmann, ISBN 1-55860-856-7. 2004

[18] Stuart J. Russell, Peter Norvig. Intelligenza Artificiale: Un Approccio Moderno. Pearson 2005

[19] Christopher M. Bishop; Pattern Recognition And Machine Learning. Springer. 2006

[20] Sutton RS, Barto AG; Reinforcement learning: An introduction. MIT press; 2018.

[21] Howard, Ronald A.; Dynamic Programming and Markov Processes. The M.I.T. Press. 1960

[22] Jens Kober, J. Andrew Bagnell, Jan Peters; Reinforcement learning in robotics: A survey: The International Journal of Robotics Research, Volume: 32 issue: 11, page(s): 1238-1274; 2013

[23] H. Zhu, J. Yu, A. Gupta, D. Shah, K. Hartikainen, A. Singh, V. Kumar,and S. Levine; "The ingredients of real-world robotic reinforcement learning," arXiv preprint arXiv:2004.12570, 2020.

[24] https://docs.docker.com/

[25] https://www.ros.org/

[26] https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html

[27] Andres, Obermeier, Sabuncu, Schaub, Rajaratnam. ROSoClingo: A ROS package for ASP-based robot control. 2013

[28] https://sir.upc.edu/projects/rostutorials/7-actions_tutorial/index.html

[29] Lentin Joseph; Mastering ROS for Robotics Programming; Packt Publishing; 2015

[30] Ramkumar Gandhinathan, Lentin Joseph; ROS Robotics Projects: Build and control robots powered by the Robot Operating System, machine learning, and virtual reality, 2nd Edition; 2019

[31] Anil Mahtani, Luis Sánchez, Enrique Fernández, Aaron Martinez; Effective Robotics Programming with ROS; Packt Publishing; 2016

[32] http://gazebosim.org/tutorials?cat=install

[33] https://www.universal-robots.com/products/ur5-robot/

[34] https://robotiq.com/products/2f85-140-adaptive-robot-gripper?ref=nav_product_new_button

[35] https://github.com/utecrobotics/ur5

[36] M. Deisenroth, G. Neumann, and J. Peters; "A survey on policy search for robotics"; Foundations and Trends in Robotics, vol. 2, no. 1-2, pp.1–142; 2013.

[37] UR5 Inverse Kinematics, Ryan Keating, Johns Hopkins University. M.E. 530.646. 2014. Updated by Noah J. Cowan. 2016

[38] https://github.com/iocchi/PLEXI

[39] Mathematical Modelling and Simulation of Human-Robot Collaboration. R.Galin, R. Meshcheryakov. DOI:10.1109/RusAutoCon49822.2020.9208040

[40] Breazeal, Cynthia and Scassellati, Brian; Robots that imitate humans.Trends in Cognitive Sciences.Elsevier. 2002

[41] S Schaal, A Ijspeert, A Billard; Computational approaches to motor learning by imitation; Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences; Volume 358; Pages 537-547; 2003

[42] S Calinon, F Guenter, A Billard; On learning, representing, and generalizing a task in a humanoid robot; IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics);Volume 37; 2007