Robot Programming and Control - Project Report

Contact: Denis.Tognolo@studenti.univr.it, Giacomo.Canella@studenti.univr.it

Denis Tognolo
VR480314
Giacomo Canella
VR481630

1 Project introduction

The aim of this project is to simulate a typical industrial task for a robot manipulator in the ROS environment. In particular we implemented a pick and place task using a UR5e manipulator, with a mlp3240 gripper mounted on it.

1.1 ROS and MoveIt introduction

The Robot Operating System (ROS) is an open source set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools. All this project use the Noetic distribution of it [1].

MoveIt [2] is a robotic manipulation platform for ROS, and incorporates the latest advances in motion planning, manipulation, 3D perception, kinematics, control, and navigation.

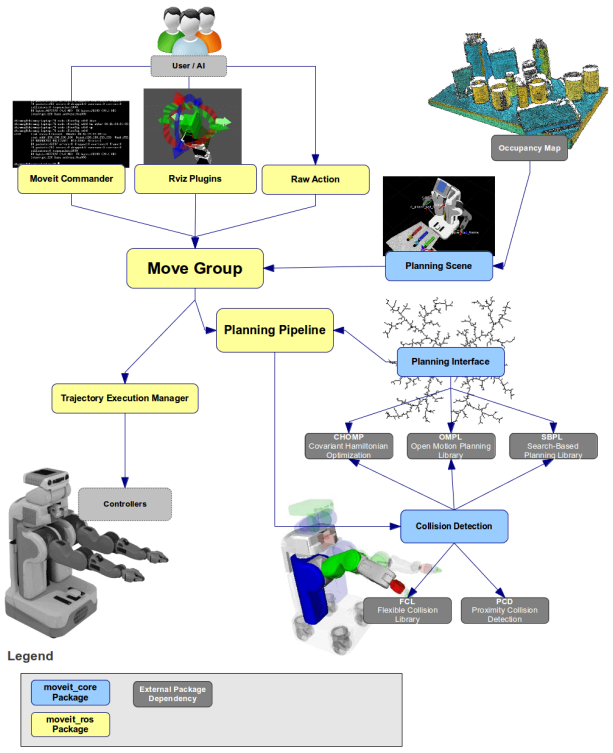

Figure 1: MoveIt Pipeline

1.2 UR5e and mlp3240 introduction

The UR5e [3] is a lightweight, adaptable collaborative industrial robot made by Universal Robot [4], that tackles medium-duty applications with a maximum payload of 5 kg, with ultimate flexibility. This manipulator is designed for seamless integration into a wide range of applications. The UR5e is one of the most popular manipulator for striking the perfect balance between size and power.

The mlp3240 is a basic two finger parallel gripper also owned by University of Verona.

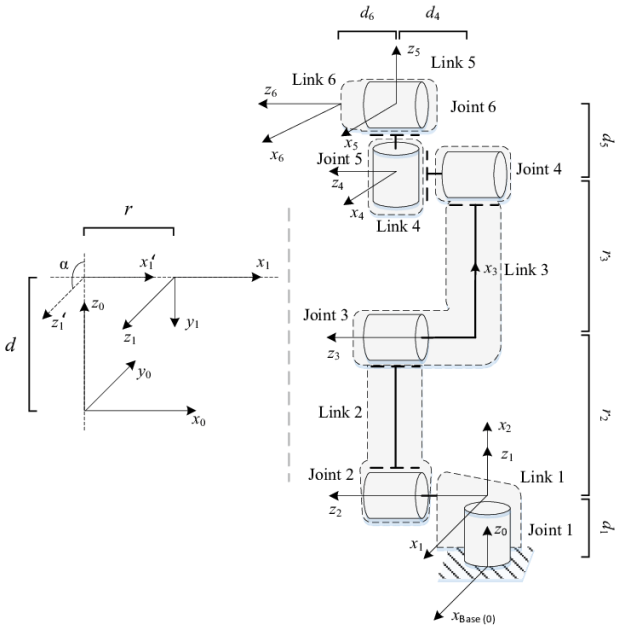

Figure 2: UR5e Denavit Hartenberg Parameters

1.3 Task and Environment introduction

The environment recreates a real world's problem. We considered as an example an industry that produces chocolate's bars and wants to implement an automated portioning system, that includes a vision system to check the quality of each product. A manipulator picks one specific chocolate bar from a shelf, choosing one of the existing 15 types of bars each positioned in a specific slot, then carries it inside a vision box where it is flipped (in

order to check the quality of the product on both sides) and finally leaves it inside a portioning machine. At this point is able to restart the task with another bar. All this task is supposed to be performed inside a safe box, even if the robot is a collaborative one, so it's safe for people by itself.

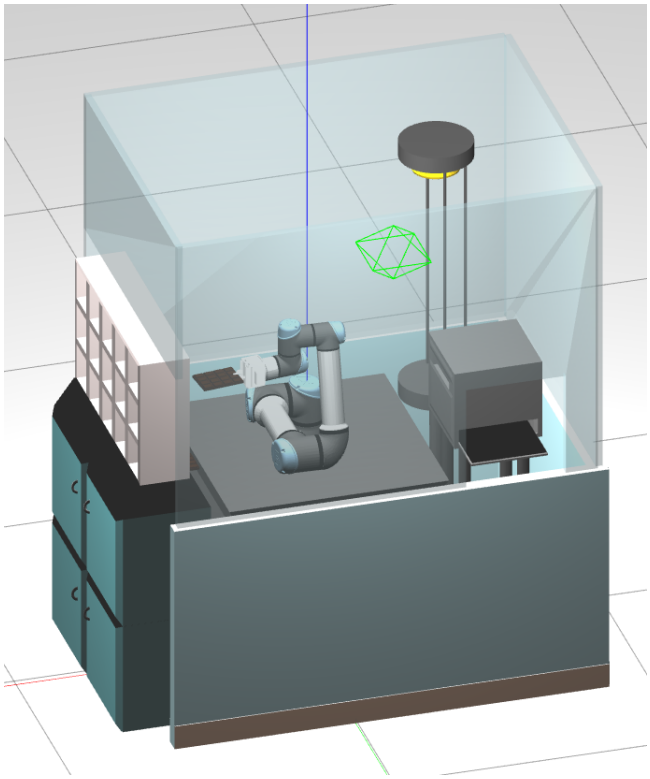

Figure 3: Portioning Line Environment

2 Package and Setup

2.1 Setup and Material sources

Regarding the UR5e manipulator we used the URDF file, containing the model description, that you can get from the official MoveIt website (`https://moveit.ros.org/robots/`) and you can find it in the universal_robot/ur_e_description/urdf folder. For the mlp3240 gripper we used the 3d model provided by other students as courtesy [5], and adapted it to the Gazebo simulation's environment. In particular we added to its URDF file some physics information to ensure better interaction with the industrial environment we have created.

To attach the gripper on the end effector of the manipulator we wrote another URDF file, that you can find in the ur5e_mlp3240_task/urdf, linking the other two. Then to spawn it inside the Gazebo environment we modified the auto-generated *gazebo.launch* file (it will be discussed later) calling our environment's file instead of the default one: *empty_world*.

The environment was created from scratch, assembling the models generated with Blender's software, in the 'portioning_line.world' file.

Concerning the URDF, we also added two plugins that we'll present you in the following sub-sections.

2.2 Gazebo Grasping Plugin Setup

Especially in older Gazebo versions, the robot often displays strange behaviours while grasping an object: the latter may wobble, explode, or fly off into space. This is because the physics engine is not optimized for grasping yet. So we used *gazebo_grasp_plugin* a package made by Jennifer Buehler that helps to overcome such issues. The main idea behind this package is that an object is detected as "grasped" as soon as two opposing forces are applied by the gripper's links on it. In this situation the object is fixed to the palm or hand link without slipping out. As soon as this condition does not hold anymore (e.g. the gripper opens), the object is detached again.

Refer to [6] for documentation.

2.3 Gazebo Mimic Joint Plugin Setup

Usually grippers have more passive joints than actives, in fact in our cases we have two prismatic joints that work symmetrically, therefore we can control one of the two and the other have to copy it. To obtain this kind of behavior we used the latest version of the Mimic Joint plugin.

Refer to [7] for documentation.

2.4 MoveIt Setup

In order to work with MoveIt's library is it necessary to build a specific package for our manipulator to setup properly all the messages and configurations required for the execution. This can be easily obtained by exploiting the MoveIt Setup Assistant [8], that can be launched through the command line *roslaunch moveit_setup_assistant setup_assistant.launch* The MoveIt Setup Assistant is a graphic user interface useful to configure any robot for the MoveIt's library. Its main function is to generate a Semantic Robot Description Format (SRDF) file for your robot. Furthermore it generates other necessary configuration's files useful for the MoveIt pipeline. The assistant only needs the URDF file of the robot with the gripper already mounted on it, and then you can start interacting with the gui: the most important parts of this procedure are the definition of the planning group (in our case one for the manipulator and one for the gripper), where you need to specify possible passive joints, and the definition of some robot's poses like the robot 'home' position or the gripper open/close configurations. Other important aspect to specify, that are almost generated automatically, are the Self-Collision matrix, the kinematic solver, and the types of controller.

After the automatic set-up, certain files require some modification: inside the config/ros_controllers.yaml file you need to set (if it has not yet been done) the type of all controller as effort_controllers/JointTrajectoryController and tune the PID for each of them. Then you also need to add a joint_state_controller to monitor your joints and this turns out to be critical for making the mimic plugin work as well. Then in the launch/gazebo.launch file you need to include your customized world file instead of the default one (empty_world) that is inside. To be more precise, we create a copy of launch/gazebo.launch file in order to leave it unaltered, and called it portioning_line.launch.

2.5 Joint Limits Setup

Another key aspect to force the robot to always perform the same trajectory, was to properly set the Joint Limits Position value. You can set it inside the config/joint_limits.yaml file.

3 Code implementation and explanation

In this section we will present you in more details the developed code for the pick and place task and our design choices.

3.1 Task and environment implementation

All the objects in the environment were created using blender and than their meshes were exported in .dae format to be included in Gazebo. You can find all models in both format (.blend and .dae) inside the models folder of our package.

The objects were modeled as described in the following list:

- **Chocolate bar**: 10x140x100 mm parallelepiped
- **Shelf**: composition of 3 rows and 5 columns of 150x150x150mm drawer
- **Vision box**: cylinder with 300 mm of radius and 950 mm of height
- **Portioning machine**: 300x300x300 mm cube with a 200x50 mm hole
- **Table**: 750x750x750mm cube

Some of these objects are provided with a support that allow them to belong to the reachable space of the manipulator.

- **Shelf support**: 900 mm
- **Vision box support**: 750 mm
- **Portioning machine support**: 850 mm

All of these geometric information are defined in the beginning of the main file, that you can find at ur5e_mlp3240_task/src/pick_and_place.cpp file, but then are completely managed by the two classes we implemented and that we are going to present in the following subsections.

It's important to highlight that all the visualizations of the objects in Gazebo refer to the ur5e_mlp3240_task/world/portioning_line.world, that are in no way related to the task code, so any kind of modification in geometries has to be done in both files.

In order to make the task more challenging we tried to limit the whole setup dimension as much as we could, we end up with a configuration of 1100x850 mm size.

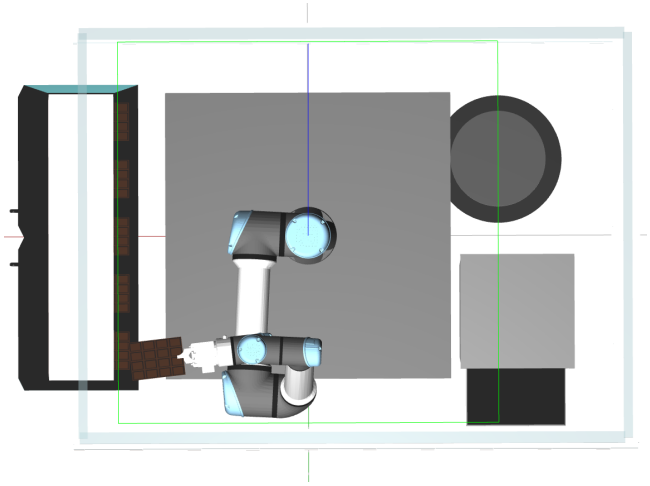

Figure 4: Portioning Line Top View

3.2 Chocolate Portioner Environment

This class class computes all the geometrical calculation required to produce the desired coordinates (the chosen chocolate bar coordinates, vision point coordinates etc...) starting from the environment spatial information.

So we implemented it by writing a cpp class called chocolate_portioner_env.h that you can find at ur5e_mlp3240_task/include folder.

For this class we implemented the following methods:

- **chocolate_portioner_env**: class constructor.
- **set_RPY**: it sets the orientation (quaternions) of a given pose starting from RPY angles in radians.
- **check_chocolate_bar_code**: it checks if the typed code exists in the inventory, given it.
- **compute_chosen_chocolate_bar_origin**: it computes the pose where a chocolate bar is located, given its code.

- **compute_chosen_chocolate_bar_pose**: it computes the pose that the robot has to assume in order to grasp the desired chocolate bar, given it's code and the desired orientation.
- **compute_chosen_chocolate_bar_pose**: it computes the pose that the robot has to assume in order to pick the chocolate bar correctly from the shelf, given the chosen chocolate bar origin and the desired orientation.
- **compute_vision_box_hole_pose**: it computes the pose that the robot has to assume in order to place the chocolate bar correctly inside the vision box, given the desired orientation.
- **compute_portioning_machine_hole_pose**: it computes the pose that the robot has to assume in order to place the chocolate bar correctly inside the portioning machine hole, given the desired orientation.
- **build_chocolate_bars_map**: it builds the map that contains the chocolate bar's origins, given the offset between the shelf and the chocolate bar origin (on x-axis), the shelf basement height and the quantity of each bars in the inventory (position.z = 0.0 correspond to the top of the shelf support).
- **compute_obstacle_primitives**: it computes the primitive solids for all the collision objects in the scene.
- **compute_obstacle_poses**: it computes the poses for all the collision objects in the scene.
- **compute_obstacle_names**: it computes the names for all the collision objects in the scene.

3.3 Ur5e Gripper Interface

This class produces the proper commands that allow the robot to move from a starting point to a specific goal position that can be both in joint or in Cartesian's space. This class also menages collision objects, in particular it adds them into the MoveIt Planning Scene, refer to the Collision avoidance subsection for further comments.

So we implemented it by writing a cpp class called called ur5e_gripper_interface.h that you can find at ur5e_mlp3240_task/include folder.

For this class we implemented the following methods, that can be divide in methods to deal with collision object, and methods to deal with geometry:

- **ur5e_gripper_interface**: class constructor.
- **add_collision_objects**: it adds to the MoveIt planning scene information about collision objects, given a list of obstacles (names, primitives and origins). This methods also setup an attached collision object given a grasped object name, that must be present in the obstacle names list.
- **remove_collision_objects**: it removes from the MoveIt planning scene all the collision objects.
- **attach_grasped_object**: it attach the grasped object to the robot.
- **detach_grasped_object**: it detach the grasped object to the robot
- **set_position**: it sets and returns a given position (leaving orientation equal to 0,0,0,0).
- **print_current_current_pose**: it prints the pose of a given link name, considering the actual robot configuration.
- **print_current_joints_config**: it prints the actual joint configuration (each joints angles/position).
- **go_to_pose**: it forces MoveIt to plan a trajectory that moves the robot in a desired pose and it performs it.
- **go_to_config**: it forces the robot to assume a known and given joint's configuration
- **actuate_one_joint**: it forces the robot to move a given joint with respect to a given angle/position.
- **move_gripper**: it forces the gripper to move to a known and desired configuration.

3.4 Pick and Place task

Given the classes that we had implemented and explained in the previous subsections, the pick and place task code results really easy and clean. At the beginning of this file, you can find at ur5e_mlp3240_task/src/pick_and_place_task.cpp, we simply specify all parameters that define our specific environment and robot setup, such as object spawn pose, their sizes and all the offset needed to calculate the goal pose that the robot will have to assume. This parameters must be coherent with the geometry information inside the ur5e_mlp3240_task/world/portioning_line.world file, otherwise the planning will not match the visual simulation.

Then we simply initialized the classes and feed them with all the previously defined datas.

Then the task is performed by a series of way points or joints commands.

1) Move to home position: it performs a joint space motion into the home configuration.

2) Open the gripper: it performs a joint space

motion into the open configuration.
3) Move the EE close to the chocolate bar: it performs a Cartesian space motion to the chocolate position.
4): Close the gripper: it performs a joint space motion into the close configuration.
5) Move the Chocolate Bar inside of the vision box: it performs a Cartesian space motion to the vision box position.
6) Rotate the Chocolate Bar: it performs a joint space motion actuating a specific joint with a specific angle.
7) Move the Chocolate Bar inside the portioning machine: it performs a Cartesian space motion to the portioning machine hole position.
8) Open the gripper: it performs a joint space motion into the open configuration.
9) Move back to home position: it performs a joint space motion into the home configuration.

In order to approach properly all the goal position we defined an offset equal to 1mm in addition to the overall dimensions of the chocolate bar outside the gripper (3/4 of the chocolate bar length).

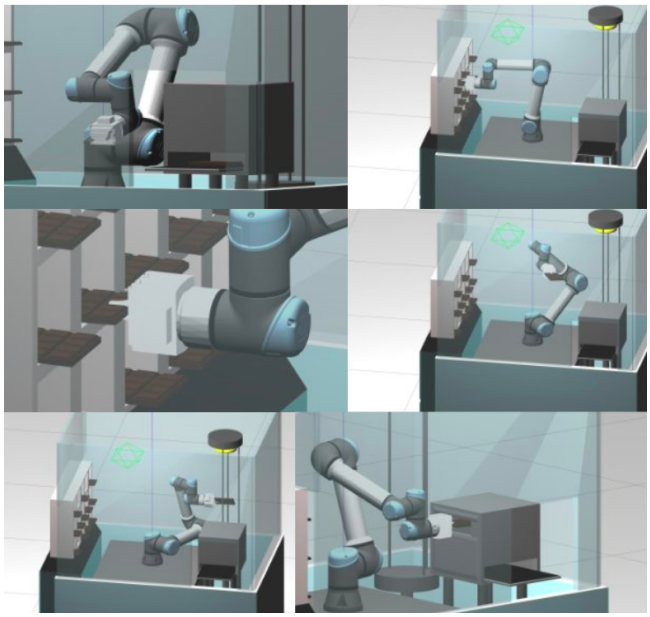

Figure 5: Gazebo_task

3.5 Collision avoidance

Placing the objects in the scene is not enough, in order to avoid collisions between the robot and the objects during the task it is necessary to add collision objects. Furthermore, once the robot picks up a chocolate bar it is necessary to add a collision to avoid troubles while moving it through the work's environment. We managed this two aspects in a slightly different way: for static object like the table, the shelf etc. we created some moveit_msgs::CollisionObject (the green ones), and for the moving one (the chocolate bar) a moveit_msgs::AttachedCollisionObject (the red one) that consist in a moveit_msgs::CollisionObject and a link name (of the robot) that will be the one at which the object will be attached. So for each object in the scene we created a "shape_msgs::SolidPrimitive" (for example a box for the shelf and a cylinder for the vision box) with the shape and pose equal to the object they had to represent. At this point, for the static objects, we proceeded by adding the SolidPrimitive variables to the scene exploiting Moveit function "applyCollisionObjects". Instead for the chocolate bar, its SolidPrimitive variable is added to it with the Moveit function "applyAttachedCollisionObject" immediately after the robot picks it from the shelf. At the and of the task it is also necessary to remove the collisions both for CollisionObject and AttachedCollisionObject so that we can relaunch the task without relaunching the simulation. All these functions that manage these processes has been implemented in chocolate_portioner_env.h while the SolidPrimitive objects has been implemented in chocolate_portioner_env.h.

Collision objects' list:

- **Shelf**: composition of SolidPrimitive Boxes.
- **Vision box**: composition SolidPrimitive Cylinders.
- **Portioning machine**: Compositions of SolidPrimitive Boxes.
- **Table**: SolidPrimitive Box of equal size to the table.
- **Walls**: two SolidPrimitive Boxes of equal size to the walls, front and back side with respect the spawn position of the robot.
- **Chocolate bar**: SolidPrimitive Box of equal size to the chocolate bar.

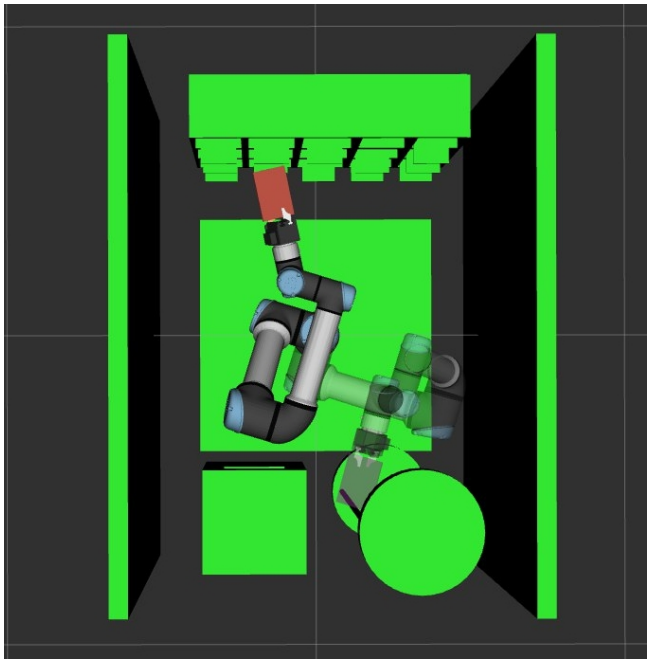

Figure 6: Collision_objects

We would like to highlight that attaching an existing moveit_msgs::CollisionObject to the moveit_msgs::AttachedCollisionObject object's field leads to some problems. We suggest you to specify the collision object directly in the moveit_msgs::AttachedCollisionObject object field, and then use this field as a new moveit_msgs::CollisionObject to be pushed in the collision_objects list, see the ur5e_gripper_interface.add_collision_objects method.

4 Execution Example

In this subsection we want to show you an example of execution. In this demo we initially inserted a wrong code, than a code that is not in the inventory for demonstrative purpose.

```
rosrun ur5e_mlp3240_task pick_and_place _code:=J2

[ INFO]: Loading robot model 'ur5e_mlp3240'...
[ INFO]: Ready to take commands for
         planning group ur5e_arm.
[ INFO]: Ready to take commands for
         planning group mlp3240_gripper.

ERROR: given code does not exist!
       Please enter another bar code [A->C, 1->5]: A4
ERROR: this bar stock out!
       Please enter another bar code: B2

Adding 36 objects as obstacles...

GOING TO HOME POSITION...
[ INFO]: Visualizing plan 1 (pose goal)

OPENING THE GRIPPER...
[ INFO]: Visualizing plan 1 (pose goal)

GOING TO THE CHOCOLATE BAR...
[ INFO]: Visualizing plan 1 (pose goal)

CLOSING THE GRIPPER...
[ INFO]: Visualizing plan 1 (pose goal)

Attaching grasped_object:B2

GOING INSIDE THE VISION BOX...
[ INFO]: Visualizing plan 1 (pose goal)

WAITING 1 SEC...
FLIPPING THE CHOCOLATE BAR...
[ INFO]: Visualizing plan 1 (pose goal)

WAITING 1 SEC...
RE-FLIPPING THE CHOCOLATE BAR...
[ INFO]: Visualizing plan 1 (pose goal)

GOING INSIDE THE PORTIONING MACHINE...
[ INFO]: Visualizing plan 1 (pose goal)

OPENING THE GRIPPER...
[ INFO]: Visualizing plan 1 (pose goal)

Detaching grasped_object:B2

GOING BACK TO HOME POSITION...
[ INFO]: Visualizing plan 1 (pose goal)

Removing 35 objects as obstacles...
```

5 Execution Disclaimer

During our several tests we have noticed that, after some consecutive runs, the simulation begins to slow down increasingly and that could cause the planner not to have enough time to find the trajectory (in particular we had problems while entering the portioning machine hole). We recommend you to flush the terminal when you notice that the simulation begins to slow down. To solve this problem we increased the Planning Time of the Planning Interface up to 100 second but we cannot guarantee that this will solve the problem for all computer (depending on its specifics).

6 Conclusions

This project proved challenging in terms of modeling a real world situation and planning an efficient task to solve a problem, but also really frustrating in terms of implementation, even knowing Ros quite well.

In fact the MoveIt library has some problems, starting from the assistant that produces a package that should be "ready for the use" but actually is not.

Furthermore, the official examples and tutorials useful to understand MoveIt's messages (necessary to command the robot) most of the time result incomplete, missing some important key points explanations. To overcame these problems we have leaned on the existing forums, but this obviously implied to face of with non-official material.

Another unpleasant part was that the MoveIt planner often produces different solutions for the same task between the various simulations, obliging us to make several tests that sometimes resulted a waste of time.

Furthermore Gazebo, whose only purpose is the simulation, turns out to have several problems in physical object interactions: a lot of aspects are not optimized.

We hope these problems have been solved in MoveIt/Ros2/Gazebo newest versions.

References

[1] ROS Noetic Ninjemys - ROS Wiki http://wiki.ros.org/noetic

[2] MoveIt Motion Planning Framework https://moveit.ros.org/

[3] UR5e - Cobot for Industrial Automation - Universal Robots https://www.universal-robots.com/it/prodotti/robot-ur5/

[4] Universal Robots: collaborative and industrial Robot — Cobot https://www.universal-robots.com/

[5] mlp3240 visual https://github.com/AntoninoParisi/rpc/tree/master/mlp3240_visual

[6] The Gazebo grasp fix plugin WIKI https://github.com/JenniferBuehler/gazebo-pkgs/wiki/The-Gazebo-grasp-fix-plugin

[7] A simple (Model) plugin for Gazebo in order to add to Gazebo the mimic joint functionality that exists in URDF (ROS). https://github.com/roboticsgroup/roboticsgroup_upatras_gazebo_plugins

[8] MoveIt Setup Assistant - ROS Documentation http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html