Approach to grasp with Romeo robot using RTM

Lluís Salord Quetglas
l.salord.quetglas@gmail.com

Technischen Universität Wien (TU Wien)
Automation and Control Institute (ACIN)

Master Thesis Project

June 17, 2016

Abstract

Contents

List of Figures

List of Tables

Acknowledgements

Chapter 1

Introduction

1.1 Motivation

A world with humanoids robots living with people is one of the dreams of any robotic researcher. Although, these robots should be able to do a wide range of tasks as humans or even better. However, a good interaction of robots with the surrounding environment, including humans, is so difficult to accomplish due to the complexity of it. There are several ways of interaction such as: (1) learning about the environment, (2) be able to move to wherever or (3) grasping all kind of object. All these fields are being studied in any robotic research institute but not all of them can implement it on real robots. Luckily for me, in the ACIN of the TU Wien, they have an humanoid robot, Romeo robot from Aldebaran Robotics, and I have chosen to use it focusing in the grasping process. Therefore, in this dissertation, it is made an approach of the grasping part using Romeo from Aldebaran Robotics. Moreover, before the grasp the robot should analyse the surroundings, recognise the object and then grasp it. Therefore, it should know how to recognise things, which is also a very important field.

1.2 Presentation of the problem

Romeo is an humanoid robot from Aldebaran Robotics company. Nao and Pepper are other humanoids from this company, these three belong to the Aldebaran's robot family, Figure 1.1, with and specific aim for each robot. On one hand, Nao is targeted to investigation, programming and learning. On the other hand, Pepper is focused in social relations with clients or users. Finally, Romeo has the aim to help old people or people who has some kind of disability.

In order to be able to achieve this aim, Romeo has to be capable of doing several tasks, which a lot of them requires to recognize objects and grasping them. Therefore, recognition and grasp objects is a huge step forward for the development of Romeo.

Currently, exists some approaches to grasp objects with several robots, but there is only one for Romeo [9], which is explained in Section 2.1, but it can not be used to achieve the main aim of Romeo. Therefore, it should be done a grasp approach in a way which consider this main goal.

Figure 1.1: Aldebaran's robot family. Pepper on the left; Romeo on the right and Nao in the middle [19].

1.3 Aim of the project

The main goal of this project is to achieve that the Romeo robot can recognise some specific objects and, if they are inside its workspace, then grasp them. It is known that this task is difficult to accomplish, due to the wide range of parts that are involve, as recognition, movement of the robot, interaction with the environment, etc. So another goal is to discern from where it comes the main error and solve it or at least propose possible solutions. Moreover, another important goal of this project is to do the task of grasping in the way of the aim of Romeo. Therefore, it has to be coherent with the idea of helping people.

Furthermore, this task is not only important for Romeo, a wide range of robots need to achieve that. Therefore, share the code and make it open source could help other research programs, and other people could help to continue and improve our work. So this work is not over when the master thesis is done, but it can be continued with any restrictions. Moreover, the code of this project should be put in a package, which should be as flexible as possible to be complemented with any kind of packages. Finally, another main goal of this project is to be explained in a way that other people can understand perfectly how it works and then improve the code.

Due to the nature of the project itself there are some limitations or requirements that it should be accomplish. Firstly, it needs to be stated that this work is a first approach to make Romeo capable to recognize and grasp objects. Due to be a master thesis, which it should be done in only a few months, and it is required some time to get used with the robot, it is probably that the final result won't be totally perfect.

Moreover, there are some software constraints which should be accomplished. On one hand, the use of an acknowledged software like ROS, explained in Section 3.2.1, so it can be easily implemented with other packages and it is a perfect framework for a open-source robotic project. On the other hand, the use of the RTM software, explained in Section 3.2.3, which is from our institute (ACIN), due to is a novel software for the recognition of objects. Finally, the packages from which this project depends should be used as they are officially. So if they need to be modify, the changes should be send to the maintainer of the package and updated.

Summarizing the main goals with their subgoals are the following:

- Approach to grasp an object recognised with the RTM software.
  - Discern by parts where comes the possible error that can be in the approach.
  - Grasp using a method coherent with the Romeo aim to help people.
- Share the work in order to help the community and also they can help us.
  - Share it in a ROS package.
  - This dissertation should explain all the concepts in a way that makes easy to understand the code and so it can be improved by the community.

1.4 Repository

One of the main ideas of this project is to be open source, so everybody can use and improve it. In order to accomplish this, all the code can be found in the following repository with also this document:

`https://github.com/lluissalord/romeo_grasper`

1.5 Structure

As the idea of this document is to help people, who want to use or improve the package, to understand all the parts, the document is structures as Chapter 3 explains some knowledge that should be known before to understand perfectly all the document. Chapter 4 is a brief summary of how it works the package and the most important if only wants to know how to use the package. Chapter 5 describes in detail all the parts of the package. In Chapter 6 are shown several experiments to see how precise is the package in any part.

Chapter 7 explains the results from the previous chapter. Chapter 8 proposes several fields to improve the package. Finally, Chapter 9 explains the conclusions of the project.

Chapter 2

Related works

2.1 Grasping by Romeo with visual servoing

Firstly, talk about a work previously done by another research group. The work [9] consist in grasping an object using visual servoing. The robot has to detect and to track with its gaze a box placed on a table in front of him, estimate the pose of the box with respect to one of its eye's camera, approach its arm near the box and then move the arm using visual feedback so that it is able to grasp the box accurately. Once this is achieved, it detects a human and delivers the box. A video demonstration can be found at [14].

In order to know where is the hand with respect to the eye's camera, the hand have a QR-Code. So firstly it moves the arm in a close position to the box using odometry, then with the detection of the QR-Code compute the pose and finally using visual servoing get the arm closer to the box.

The approach proposed, grasping with visual servoing, is a good initial approach, but it is not appropriate to achieve the main Romeo's goal of helping old people or people who has some kind of disability. Firstly, it only can accomplish the grasp with one arm, and, in a complex environment, probably it should be necessary to use both arms. Secondly, to know where is the hand it needs a specific position for the head, looking at the hand, and then track it all the time. However, Romeo is supposed to have a good interaction with humans, but this features doesn't help it.

2.2 MoveIt simple grasps

On the ROS environment exists some packages with the aim of generate grasps, one of them and very useful to do a first approach to grasping is the `moveit_simple_grasps` package. This one is thought to be used with simple object as blocks or cylinders. The package generate a lot of potentials grasps taking as an input position and orientation of the object. Firstly, is needed a configuration file that describes the geometry of the robot's end effector. Currently the tested robots are Baxter and REEM, but there are some more of the mentioned configuration files for others robots such as Romeo, Pepper, Nao and Clam. For Romeo the configuration file is named `romeo_grasp_data.yaml`, but the most recent version of the file is in this repository [20]. It should be stated that this one has been designed to use the left arm to made a side grasp and the right arm for a top grasp.

Finally, the way of generate grasps for this package is trying all the directions, with a given resolution, in the plains made by the X-Z axis and by the Y-Z axis. The source of the official package with this and more information can be found at [10].

2.3 AGILE grasp

AGILE (**A**ntipodal **G**rasp **I**dentification and **LE**arning) grasp is a ROS package [33] which localize antipodal grasps[1] in 3D point clouds. This work [34] proposes a new approach to using machine learning to detect grasp poses on novel objects presented in clutter. The input to the algorithm is a point cloud and the geometric parameters of the robot hand. The output is a set of hand poses that are expected to be good grasps. According to [34], the grasp success rate with this algorithm is 87.8% for objects presented in isolation and 73% for objects presented in clutter.

Firstly, it is used the geometry information to reduce the size of the sample space. In order to do that the grasps have to fulfil the condition of the hand must be collision -free and part of the object surface must be contained between the two fingers. Secondly, the geometry is also used to automatically label the training set. In order to label a subset of grasp hypotheses is used the condition of having an antipodal configuration. Therefore, a large amount of training data can be used and so improving the performance.

[1] In an antipodal grasp, the robot hand is able to apply opposite and co-linear forces at two points where the line connecting the contact points lies inside both friction cones [34].

2.4 Romeo MoveIt Config

At some point is necessary to use an interface and simulations to work with a robot. An option to do this, using the MoveIt platform, is the `romeo_moveit_config`. This package gives the necessary configuration files and launchers to use MoveIt with Romeo. Therefore, it allow to launch Rviz with a model of Romeo and interact with it. This model can represent a simulation or a real one which should be connected to ROS.

There are two main launch files: (1) `demo.launch`, which run Rviz with MoveIt and a Romeo model but executing a fake joint control, so doesn't move the real joints; (2) `moveit_planner.launch`, the same as before but this time it moves the real joints. Inside both files it also launchs the `move_group` class which allows to use a planner for the movements of some groups of links and joints. This class is explained with more details in the next section.

2.5 Romeo MoveIt Actions

As the last one, the `romeo_moveit_action` package allow to use `moveit_simple_grasps` with Romeo. Moreover, it also use the `move_group` class to make an easier use of high level commands such as reach to pose, pick or place at a pose, among others. In Figure 2.1 can be seen the relation between these packages and their main functions. Therefore, the `romeo_moveit_action` package should allow the user to control Romeo to do tasks as picking or placing.

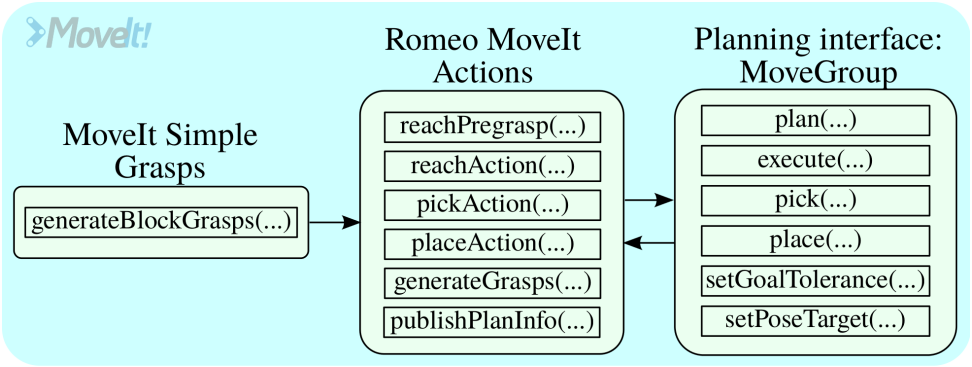


Figure 2.1: Main functions of the necessary packages for Romeo MoveIt Actions

Firstly, before start to describe `romeo_moveit_action`, is necessary to explain a bit the `move_group` class. This is the class in MoveIt that controls the current state, the target state, the planner and the movement of a group of links and joints. These groups are defined in an specific `.srdf` file in `romeo_moveit_config`. Usually, to work with a

robot like this, with 7 DoF, is used the `left_arm` or `right_arm` group with `left_hand` or `right_hand` group as end effector, this is explained in Section 3.1.1.

In order to be able to make plans and execute the movements, the `move_group` class is working together with the simulator Rviz and the pertinent solver. Moreover, can be set a tolerance for the goal and the target state of the plan can be set as a: (1) pose on a reference or (2) joint state.

Referring to `romeo_moveit_action` package, the most important part is the planning and execution of the grasping. Although, there are two ways of doing that:

- Using `graspPlan` function, Figure 2.3.
- Using `reachAction` function, Figure 2.4.

The main differences between one and other are:

1. How to calculate the target pose
   (a) `reachAction` use the geometry of the gripper given by `romeo_moveit_config` and put the target pose in a way that the gripper will be close to the object applying the offset of the `romeo_grasp_data.yaml` from `moveit_simple_grasps`.
   (b) `graspPlan` use `moveit_simple_grasps` to generate all the possible grasps for the object and choose one of them to make a plan. Therefore, this way, also using the `romeo_grasp_data.yaml` file, but it adapts to the orientation of the object.
2. Attempts and approximate solutions
   (a) `reachAction` first try with different tolerances till the plan succeeded or reach the maximum attempts. In this last case, try to get an approximate solution.
   (b) `graspPlan` only try with one tolerance, but, as said before, it tries a lot of different grasps.

Once the gripper is at the right position is time to make the picking. Therefore, is used the `pickAction` function to pick the object, Figure 2.5. In this case is used again the grasps generated by `moveit_simple_grasps`, but this time in the picking stage. This function executes automatically the movement when, after testing all the pickings, there is at least one that works.

Furthermore, there is another interesting function named `publishPlanInfo` which publish in `/trajectory` topic the planned trajectory and the last position of the plan. Moreover, it calculates the distance between target position and the final position of the plan.

Although, the trajectory is in the joint space, using the forward kinematics can know the position of the final position. Finally, a summary of the topics and services used and provided by `romeo_moveit_actions` is showed in Figure 2.2.

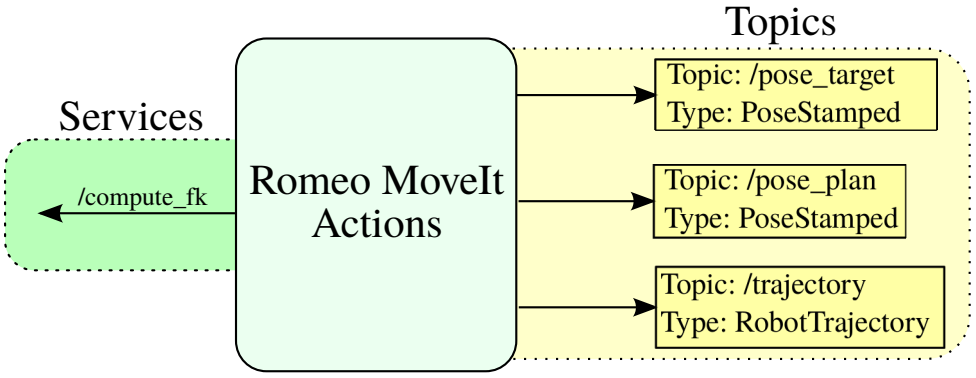


Figure 2.2: Topics and service of the package `romeo_moveit_actions`

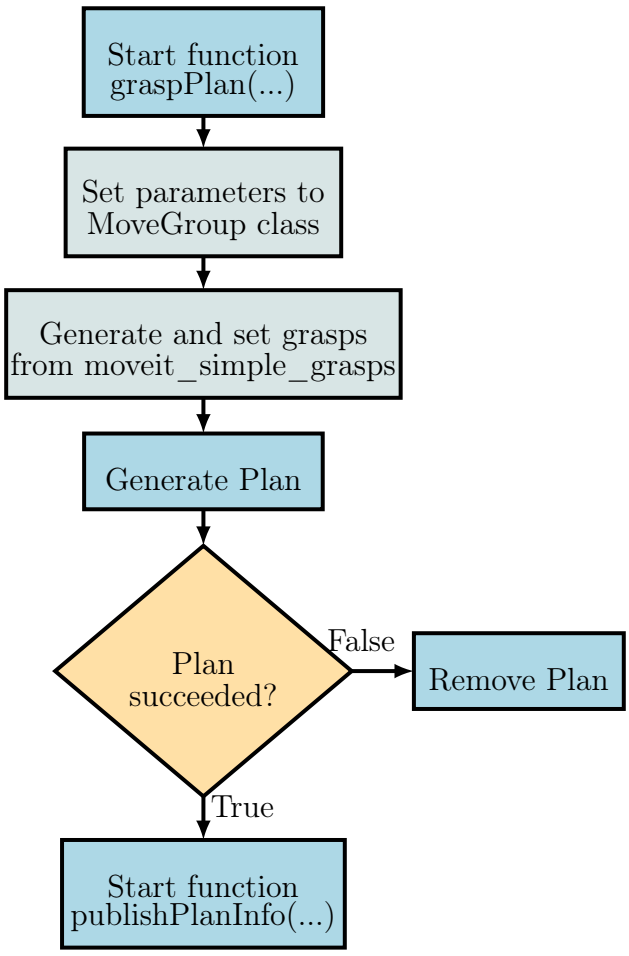


Figure 2.3: Flow sequence for function `graspPlan(...)`

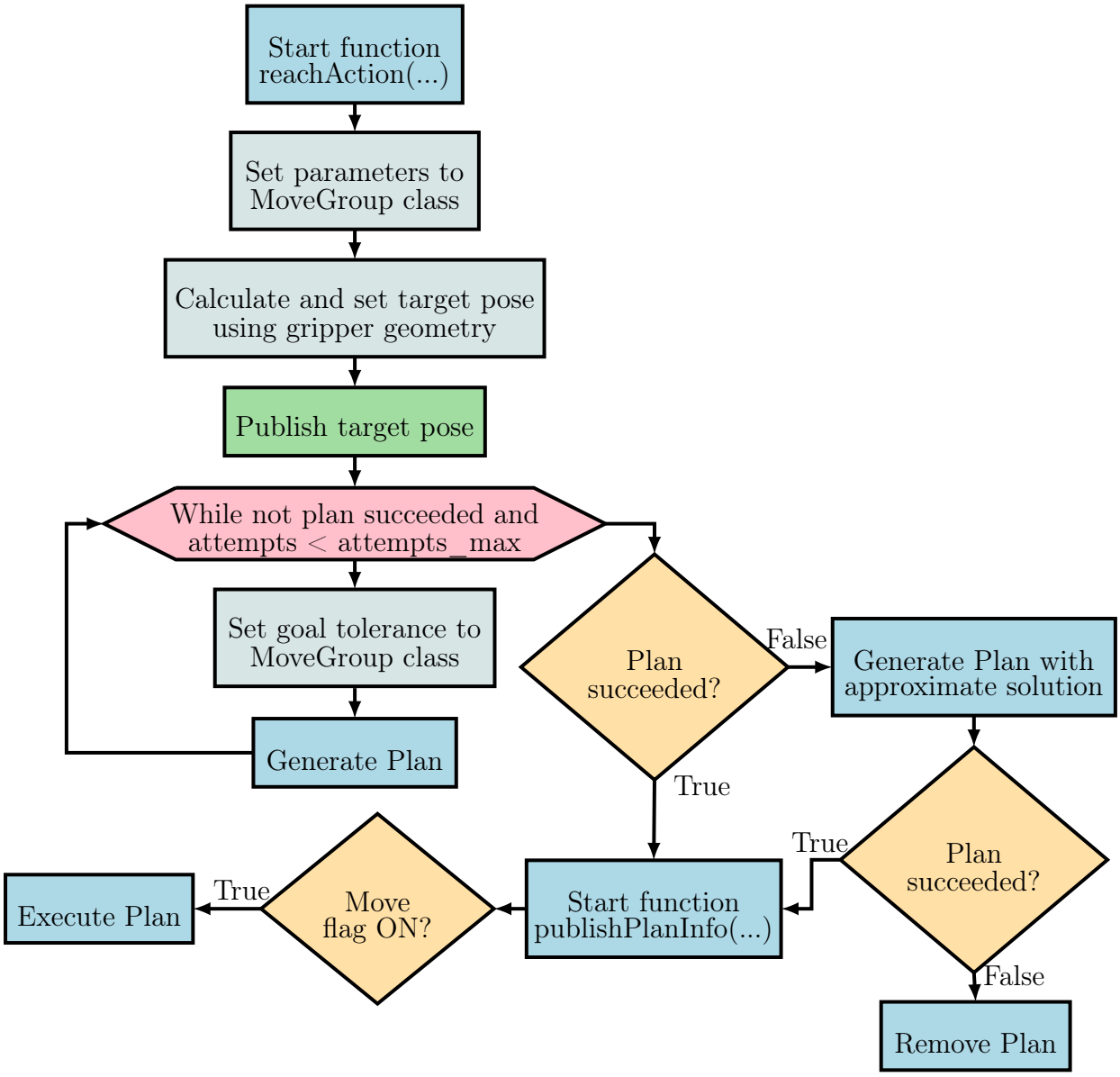


Figure 2.4: Flow sequence for function `reachAction(...)`

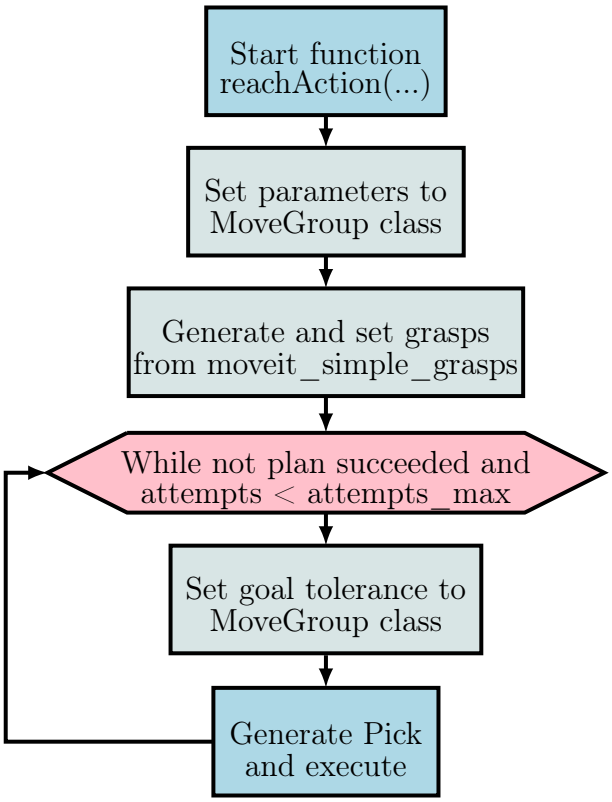


Figure 2.5: Flow sequence for function `pickAction(...)`

Chapter 3

Robot and camera system

3.1 Romeo robot

A said previously, Romeo has the aim to help old people or people who has some kind of disability. Therefore, Romeo has been build with an structure which allows it to open doors, climb stairs and reach objects on a table. Moreover, it has several sensors which allow him to understand the environment and adapt him to it. Although, in order to make a good use of this hardware, it is necessary to complement it with a powerful software as is NAOqi.

Figure 3.1: Romeo robot [22]

3.1.1 Hardware

Because of the aim of this package, this section is focused on the hardware related to: (1) the arms, in order to do the grasp and (2) the 3D vision of the robot, due to the object recognition and positioning.

Romeo arms

Each Romeo arm has 7 DoF plus 1 DoF to open and close the hand, so it has one redundancy for the movement of the arm. The links of the left arm are shown in Figure 3.2, the right arm is symmetric and changing the initial "L" or "l" to "R" or "r".

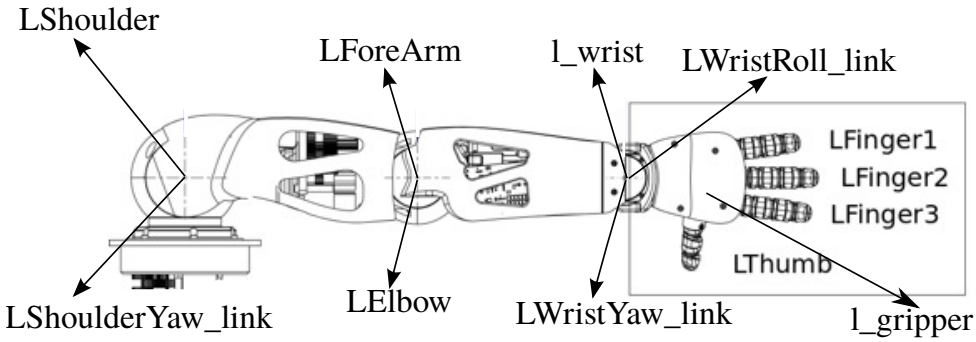


Figure 3.2: Links of Romeo left arm [1]

Although, an arm can be seen as an entire group, in robotics is usually to split it in two groups: (1) the arm group which is the main part of the arm and (2) the end effector group which is the gripper with the joints that allow to interact with objects. In order to understand the tree of links-joints and the distribution of this groups in Romeo there is the Table 3.1 which shows the parent and child link of every joint and the group which it belongs. It should be stated that when is requested to move a group to a position, is used the position of the last link of the group. Furthermore, the `L/RHand` joint is special because it has the aim of open and close the hand moving all the fingers together. Moreover, there is a group for every arm, but not used any more, named `arm_hand_left` or `arm_hand_right` which takes all the joints and links of the arm and hand together.

Group	**Joint**	**Links**	
		Parent	**Child**
left_arm	LShoulderPitch	torso	LShoulder
	LShoulderYaw	LShoulder	LShoulderYaw_link
	LElbowRoll	LShoulderYaw_link	LForeArm
	LElbowYaw	LForeArm	LElbow
	LWristRoll	LElbow	LWristRoll_link
	LWristYaw	LWristRoll_link	LWristYaw_link
left_hand	LWristPitch	LWristYaw_link	l_wrist
	LHand	l_wrist	l_gripper

Table 3.1: Joints of Romeo left arm with parent and child links

Romeo 3D vision hardware

Romeo has several vision sensors: (1) one RGB camera on each eye; (2) two more fixes above each eye and (3) an optional ASUS Xtion on the cap. Therefore, Romeo is capable of having depth data with the RGB cameras, however, some functionalities are only available with the ASUS Xtion. When this 3D sensor is connected to the robot, all

the functionalities that use depth data change the source and use the information from the Xtion camera.

Referring to the ASUS Xtion camera, according to [1], it has a color camera of 0.3 Mp and a depth camera up to 320x240 at 20 fps. Besides, the focus range is from 80 cm to 3.5 m and the field of view is 58º in horizontal and 45º in vertical.

3.1.2 NAOqi

NAOqi is the name of the main software that runs on the robots control it. The NAOqi framework is `Cross plataform` and `Cross language`. Therefore, it is possible to develop in Windows, Linux or Mac and using C++ or Python. Although, below it is only explained a brief description about NAOqi proxies and NAOqi modules, more information about the framework can be found at [1].

On one hand, the NAOqi modules are classes within libraries that contain various methods. Every module has a particular subject which it is controlling. On the other hand, the NAOqi proxies is an object that has the behaviour of the NAOqi module that represents. Therefore, the proxy from a module contain the methods of the pertinent module.

Because of the aim of this project there are four main modules:

- `ALMemory` which handle all the key information related to the hardware configuration
- `DCM` in charge of the communication with almost every electronic device in the robot except sound and cameras.
- `ALMotion` in charge of the movement of the robot
- `ALVideoDevice` which provides images from the video sources in an efficient way.

Below is explained the most important things about this modules just to have an idea and to be able to understand the following Section 3.2.2.

DCM

Firstly, the DCM module, it is the link between NAOqi modules and the software in electronic boards. Therefore, modules like ALMotion use the DCM to send commands to Actuators, and also get sensor information returned by the DCM in ALMemory. Therefore, using directly DCM commands to control the robot is faster and more efficient, but

is more difficult to use than the NAOqi modules. An overview of how the DCM works can be seen in Figure 3.3.

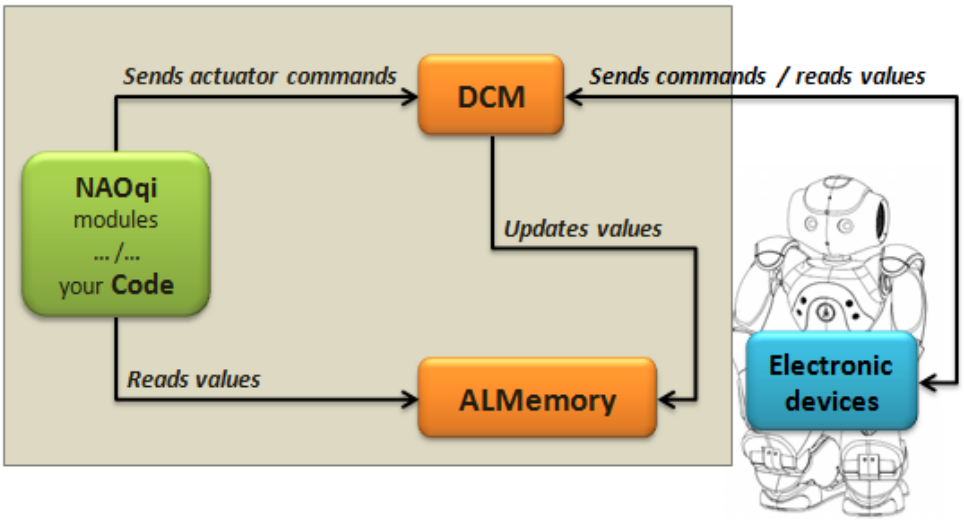


Figure 3.3: Behaviour of DCM module [1]

ALMotion

Secondly, the ALMotion module, it can do both get information from joints and set specific angles or trajectories. Some of the most important functions, at least for this work, are the following ones:

- `getStiffness()` and `setStiffness` to get or set the values of stiffness to one or more joints.
- `getAngles()` and `setAngles` to get or set the angle to one or more joints. Moreover, can set for every joint the fraction of maximum speed use to reach the specific angle.
- `angleInterpolation()` to set an interpolated trajectory for one or more joints, given the angles for specific moments.

ALVideoDevice

Finally, the ALVideoDevice module, it allows the package to get images from the sources available from the robot. As said in Section 3.1.1, there are several cameras, so the images from one or other camera can be got depending on the camera index. For this module, there are two main functions:

- `subscribeCamera()` given the features of the desired image (source, resolution, colorspace and fps) returns the string handle which identify the subscriber.

- `getImageRemote()` given the string handle returns the container of the latest image from the video source.

3.2 Software required

3.2.1 ROS (Robotic Operative System)

***R**obot **O**perating **S**ystem* [27] is an open-source and flexible framework to develop robot software. There are five main elements in this framework: (1) nodes which are the process; (2) messages that are used for the communication between nodes; (3) topics that is where the message is published by the node; (4) services is the where the node have to send a request message to get a response message and (5) actions which are like a service but can send a feedback while the activity is not still done. ROS allow the communication between process in two different ways: (1) publisher/subscriber system and using services.

1. Publisher/subscriber system is anonymous and asynchronous. Moreover, can be more than one publisher and/or subscriber for an unique topic.

2. Services are defined by two messages: (1) request message from the node and (2) response message. Therefore, the response message is given by the node which allocate the service depending on the information given in the request message.

Although, the main aim is to be able to reuse the code yet develop in robotics investigations. Also the philosophical goals of ROS can be summarized [26]: (1) Peer-to-Peer, (2) tools-based, (3) multi-lingual, (4) thin, (5) free and open-source.

ROS is a required software for `romeo_grasper` because this is the framework used to communicate the process in charge with the different devices and Romeo. Therefore, it is needed for the robot and for every device a way to communicate with ROS.

3.2.2 Ros packages for Romeo

Aldebaran company has created some packages to communicate Romeo, Nao and Pepper with ROS. At this stage are only explained some which are related with Romeo, but some of them are used also for Nao and Pepper. Although, these packages are open-source, so are continuously improving them by the community, but before the changes are released as oficial, their are revised by experimented developers on Romeo.

On one hand, there are the packages used for the three robots that are in the `ros-naoqi` repository [6]:

- `naoqi_bridge` metapackage which include the following packages:
  - `naoqi_driver_py` Python code in charge of getting data from joints.
  - `naoqi_pose` which handle the movement of the robot.
  - `naoqi_sensor_py` get the information from sensors as camera, contact sensors, microphone and sonar.
  - `naoqi_tools` allow to generate and modify Aldebaran's robot models easily.
- `naoqi_dirver` another version of `naoqi_driver_py` in C++, but less developed
- `naoqi_bridge_msgs` contains all the needed messages, services and actions to use the naoqi packages.

On the other hand, the packages used from `ros-aldebaran` repository [5] are the following:

- `romeo_robot` metapackage which include the following packages:
  - `romeo_bringup` used to launch the essentials nodes of the metapackage.
  - `romeo_dcm` which handle the communication with the Romeo electronic devices.
  - `romeo_description` contains the model of the robot in different type of files
  - `romeo_sensors_py` add the depth camera and reuse `naoqi_sensor_py`.
- `romeo_moveit_config` explained in Section 2.4.
- `romeo_moveit_actions` explained in Section 2.5.

As described in Section 3.1.2, NAOqi proxies are used to connect to a NAOqi module. Therefore, the base of all these packages is to create the pertinent proxies to send data to NAOqi or to get data from NAOqi and then publish it on ROS. This way of working is shown in Figure 3.4. Below are explained the most important packages to work with `romeo_grasper` package.

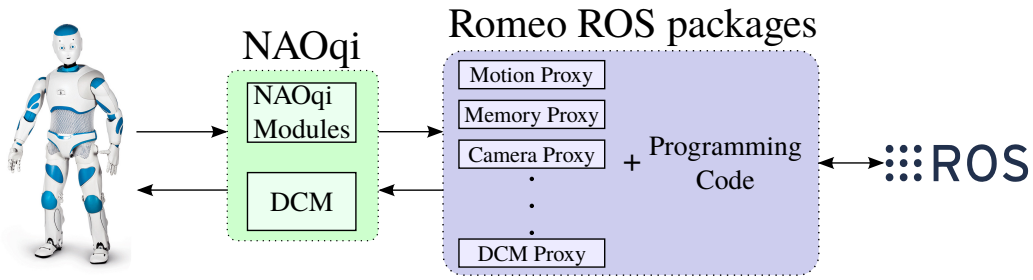


Figure 3.4: Communication between Romeo and ROS

NAOqi driver

As said previously, there are two different NAOqi driver packages: (1) `naoqi_driver_py` and (2) `naoqi_driver`. Due to unknown reason `naoqi_driver_py` is more developed than `naoqi_driver`, so it is better to work with the first one. In order to get the information from the robot joints, this package is connected to ALMotion module. Then, it gets and publish this data to ROS with a frequency of 25 Hz. As shown in Figure 3.5, the topics where it is published are `/joint_states` and `/joint_stiffness`.

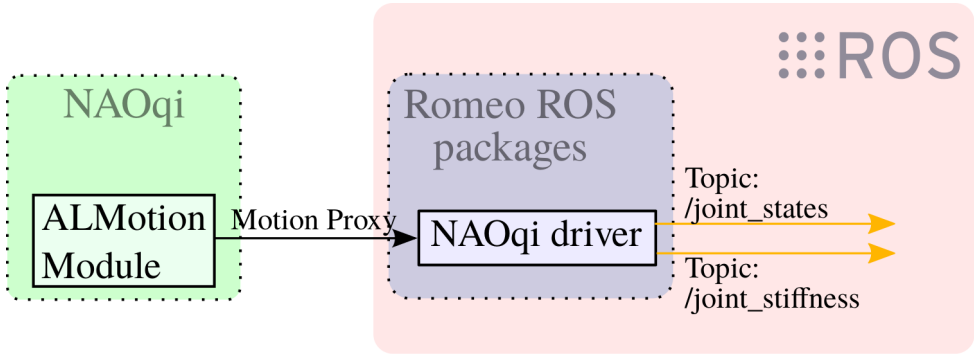


Figure 3.5: Overview NAOqi driver package

NAOqi pose

Although, this package uses two nodes: (1) pose controller node and (2) pose manager node; only pose controller node is important for `romeo_grasper`. Whereas the pose controller node is in charge of sending the goal angle or goal trajectory to the joints, pose manager only is used to move the joints to a predefined posture. Therefore, below is only explained the pose controller node.

Therefore, using the motion proxy it sends the pertinent commands to ALMotion module depending on from where the pose controller node take the information. There are several ways to communicate with this node, as can be seen in Figure 3.6. Among these, the one used in `romeo_grasper` is the action `/joint_trajectory`. In order to run this

action the request message has to contain the angles of the joint for each specific time from start. Then, the ALMotion function `angleInterpolation` is run in another thread and when the action is done it sends the final joint state.

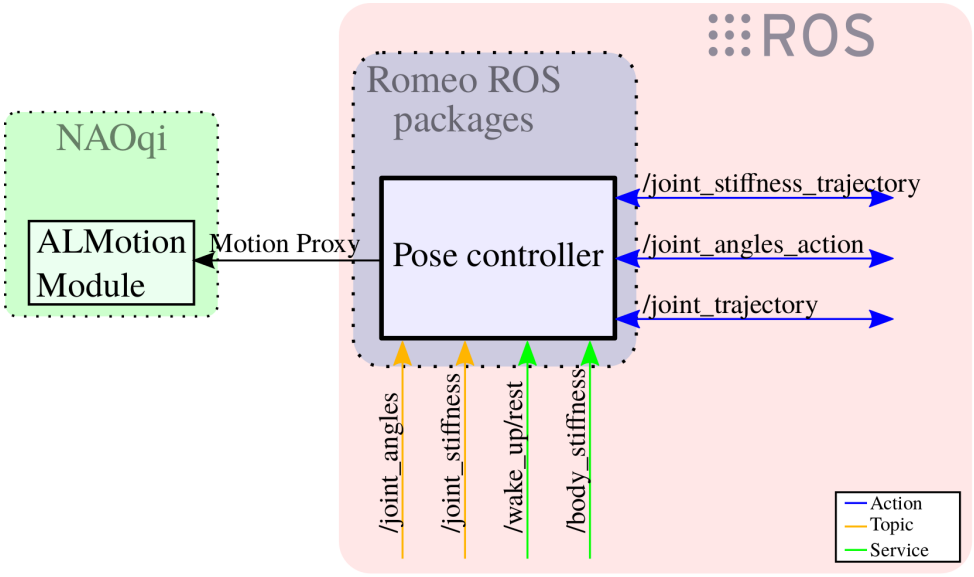


Figure 3.6: Overview Pose controller node

DCM driver

Although, previously is said that the DCM handle the communication with the Romeo electronic devices, till the moment of writing this, the code developed only implements the communication with the Romeo joints to set the pertinent angle at the precise moment. Despite the fact that the `naoqi_pose` can also do this, using DCM is more efficient and more direct, as explained in Section 3.1.2.

In robotics is needed to make a realtime control of the joints, in this case to do that is used `controller manager` from the `ROS Control` metapackage. Therefore, when someone or a package wants to send commands through DCM should communicate with `controller manager` and then this send the commands to `DCM driver` in the right way to have realtime control. An overview of this communication is shown in Figure 3.7. Moreover, in Figure 3.8 can be seen how it works this package.

Romeo sensor

Because of `romeo_sensor_py` is an addition of `naoqi_sensor_py`, it should be understood how works this last one. Firstly, initialize the ROS publishers and the camera proxy. Secondly, it gets and applies the camera parameters. After that, the function

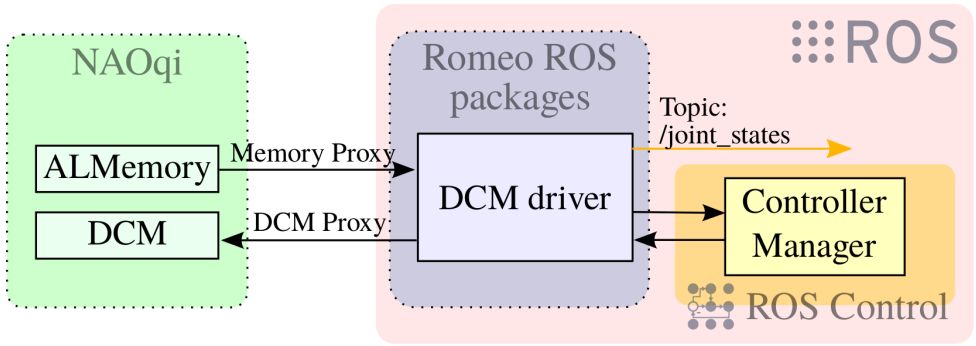


Figure 3.7: Overview Romeo DCM package

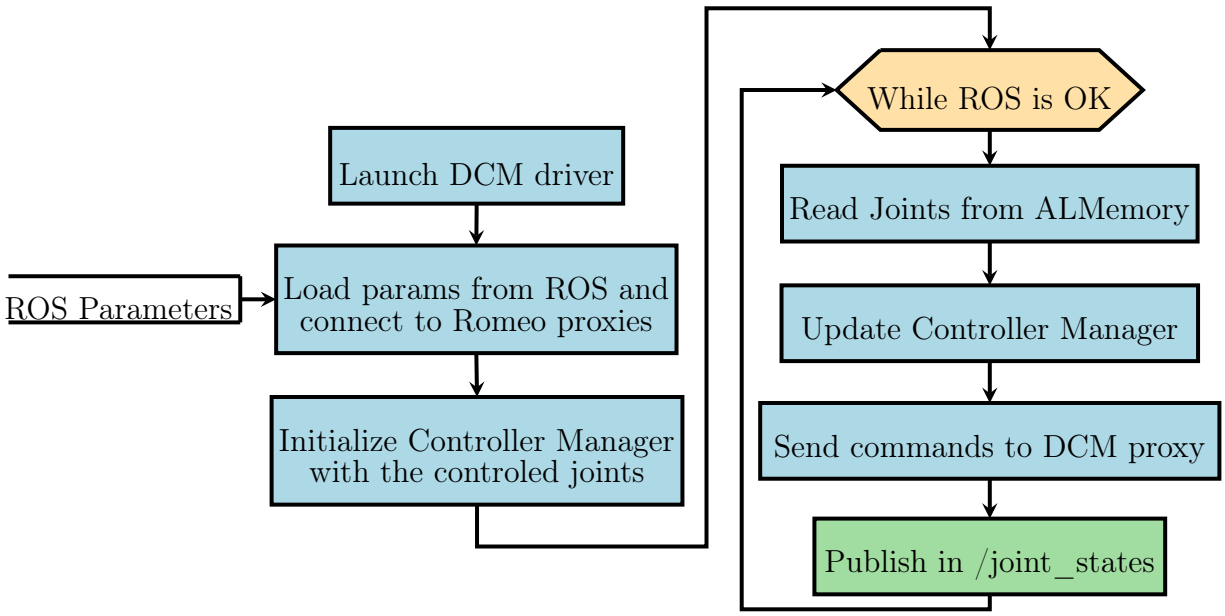
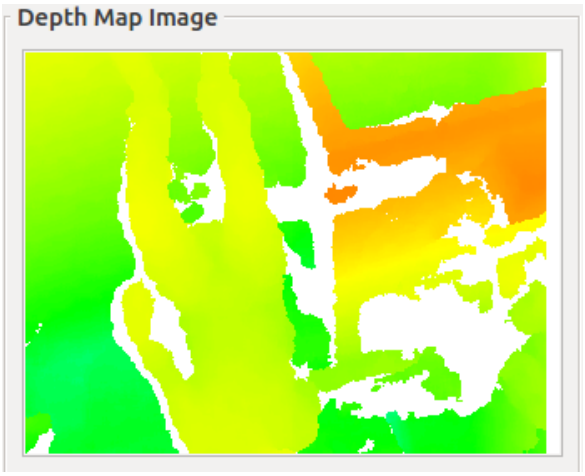


Figure 3.8: Flow sequence of Romeo DCM package

`subscribeCamera` is used to then can get the image container. Finally, it transforms all the received data to image messages to publish in the pertinent ROS topic.

In order to add the depth camera is used the same code structure as in `naoqi_sensor_py`, but changing the camera index and adapting the image type to ROS. Unfortunately, it seems that this part is not well implemented, because at least with the test done here it gets a dark image, this fact is shown in Figure 3.9.

Looking in the code one tested idea to solve it has been to change the camera index which currently is 2 (names `kDepthCamera`) which, according to [1], to connect to ASUS Xtion should be 4 (named `kXTION`). After a lot of tests, changing the encoding and other features, it hasn't reach any good result.



(a) Image from Monitor, which is Aldebaran official program to see the what are the cameras recording.

(b) Image from the topic where `romeo_sensor` publish.

Figure 3.9: Comparative images from the depth camera.

3.2.3 RTM (Recognition, Tracking and Modelling of Objects)

RTM-Toolbox (Recognition, Tracking and Modelling of Object) [25] which is a software part of a global package named V4R (Vision for Robots). The RTM-Toolbox includes, as its names says, the three features: recognition, tracking and modelling of objects. However, concerning to this project, two of them are the most interesting: (1) RGB-D object modelling for object recognition and tracking and (2) real-time object pose tracking.

On one hand, RTM-Toolbox includes a flexible system to reconstruct 3D models of objects captured with an RGB-D sensor. A major advantage of the method is that unlike other modelling tools, this reconstruction pipeline allows the user to acquire a full 3D model of the object. This is achieved by acquiring several partial 3D models in different sessions that are automatically merged together to reconstruct a full 3D model.

On the other hand, the real-time object pose tracking [23] [7], combines complementary interest points, for textured objects and for uniformly colored objects, in a common tracking framework which allows to handle a broad variety of objects regardless of their appearance and shape. Then, the point cloud around the detected object is analysed to confirm that the depth of the object is the right one.

Although, the V4R library is independent from ROS, there are wrappers for ROS [32]. Therefore, the features described above can be used though ROS. However, there are some limitations. Even using a RGB-D sensor, to run RTM-Toolbox with the modelling tool is necessary to have the sensor launched though OpenNi. The problem appears with some

RGB-D cameras that are not supported by OpenNi like the Intel RealSense.[1] Therefore, it cannot use the RTM-Toolbox with the modelling tool and only can use this application the Microsoft Kinect and ASUS Xtion cameras. Except for this, in all the others features is possible to use any camera which can publish PointsClouds in a topic, see Figure 3.10.

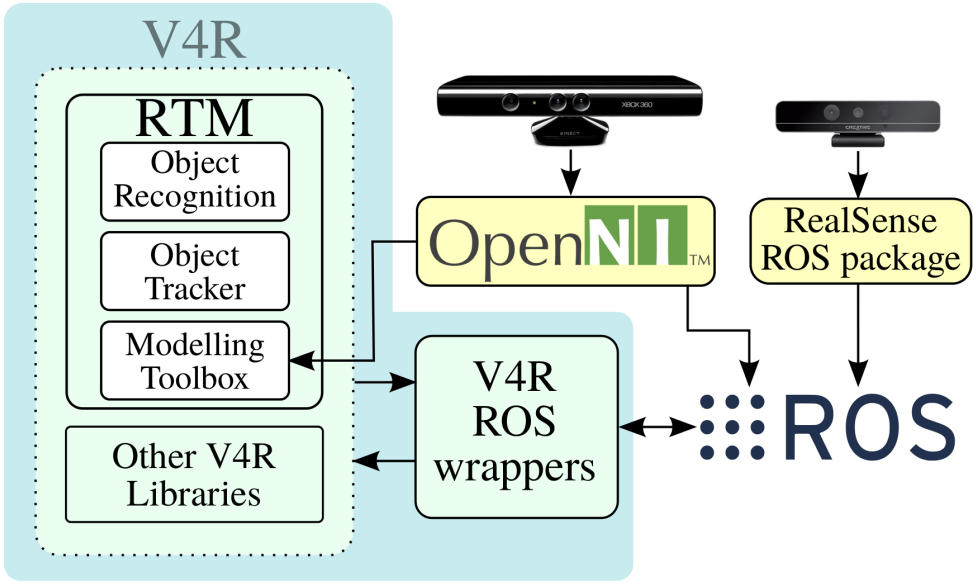


Figure 3.10: Communication between V4R - ROS - Cameras RGB-D

In order to use the toolbox and the tracker in [32] is explained the instructions to install and a brief tutorial. However, it is good to add some information. Firstly, the way to be able to make a full 3D model of an object, for this is needed to use the multi-session modelling as explained in [24]. Secondly, If the object itself is texture-less, the software relies on background texture in order to successfully model these kind of objects. So a good option is to add a textured sheet of paper on the supporting surface. Thirdly, the origin of the modeled object is posed automatically by the toolbox, so probably is not on the center of the object which is needed for the grasping process. Finally, by default the tracker is looking for the camera topic in the `/camera/depth_registered` namespace. Although, OpenNi use this namespace, the RealSense ROS package uses `/camera/depth`.[2] So to change it is needed to use the prefix `-t NAMESPACE` when the service is started as the following example:

```
rosrun object_tracker object_tracker_service -m MODEL_PATH -t /camera/depth
```

Referring to the tracker and how to interact with it, see Figure 3.11. Firstly, it should be launched with a command like the one above, then if it finds the `.../camera_info` topic and the model the service `start_recording` can be called. After that, the broker is looking for the modeled object and if it has enough confidence publish a message with

[1] Installation of the camera drivers explained in Appendix A.3

[2] RealSense-ROS doesn't have support for `/camera/depth_registered` by now, but is in process [16].

the pose and the confidence in `/object_tracker/object_pose`. Moreover, the tracker is publishing constantly images of the camera with the axis at the origin of the object, if is found, and with a bar representing the confidence in `/object_tracker/debug`. Finally, in order to change the model use to track, it is needed to call the `change_tracking_model` service with the path of the new model and after that start the recording again.

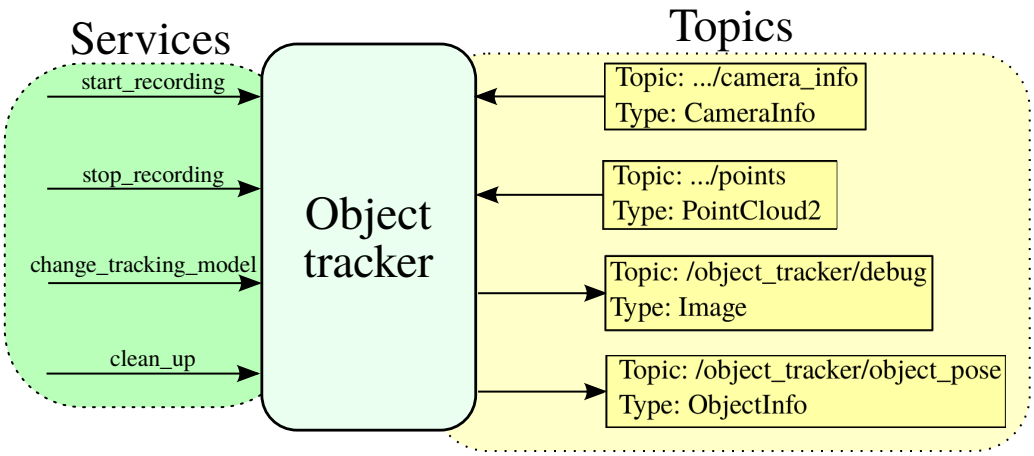


Figure 3.11: Topics and services of the module Object Tracker

It should be stated that the way to identify an object is done transforming the point clouds to an image and then from this are extracted a kind of SIFT features which describe the object. Then, the depth information is used to reaffirm that is the right object.

3.2.4 Inverse kinematic solvers

For solving inverse kinematics problems there are two main approaches: (1) numerical solution and (2) analytical (closed forms) solutions.

On one hand, *KDL* (Kinematics and Dynamics Library)[29] is a library for computing forward and inverse kinematic queries with numeric recursive solving. This library is included in the MoveIt packages and it is the default solver if another one is not specified. However, it might be very slow or trap in local maximums due to recursive nature of numerical solutions for solving IK problems.

On the other hand, *OpenRAVE* (Open Robotics Automation Virtual Environment)[12] is a software for simulating and deploying planning algorithm. It provides a tool called IKFast which can generate C++ code (independent of any library) representing the model of the manipulator for solving forward and inverse kinematic queries. The generated code is able to solve the request on the order of 4 microseconds [11]. In order to generate the C++ code for the robot, it should be provided a Collada model [17] of the robot. Besides, as stated in Section 3.1.1, the arms have one redundancy so for the C++ code generation

is needed to specify which should be the free joint. Therefore, to solve IK queries with IKFast some of the key issues is to find proper values for this free joint and which joint should be selected.

Chapter 4

Design of Romeo grasper package

The next stage introduces the general idea of the package here exposed. After this chapter, on the Implementation Chapter 5, the package is explained in detail.

Before the overview of the package, it should be stated that the purpose of `romeo_grasper` is to combine the different dependencies to achieve that Romeo grasps an object recognised by the camera. There are several dependencies, such as: (1) ROS packages which communicates with Romeo; (2) simulator and kinematic solvers to move the robot to the desired position; (3) ROS packages which generates the grasps; (4) camera drivers and (5) RTM (Recognition, Tracking and Modelling of Objects) software which allow to know where is the object respect to the camera.

A brief overview of this package can be understood by the Figure 4.1. First of all, the position of the object is got with the camera and RTM software. Then is time to compute a pre-grasping trajectory and a picking trajectory using inverse kinematics. The next stage is using the simulation to test these trajectories. After that, we send the joint commands to NAOqi through the Romeo ROS packages to finally move the robot joints and make the grasp.

4.1 Get position of the object

In order to be able of grasping object which has been recognized by the camera, we need the position of the object on the robot reference. This position is got by the camera which should be positioned too. The ideal case would be using the Romeo RGB-D camera mentioned in Section 3.1.1, because the relative position of the camera from the robot is known by the `romeo_description` package and we don't need extra connections. But is

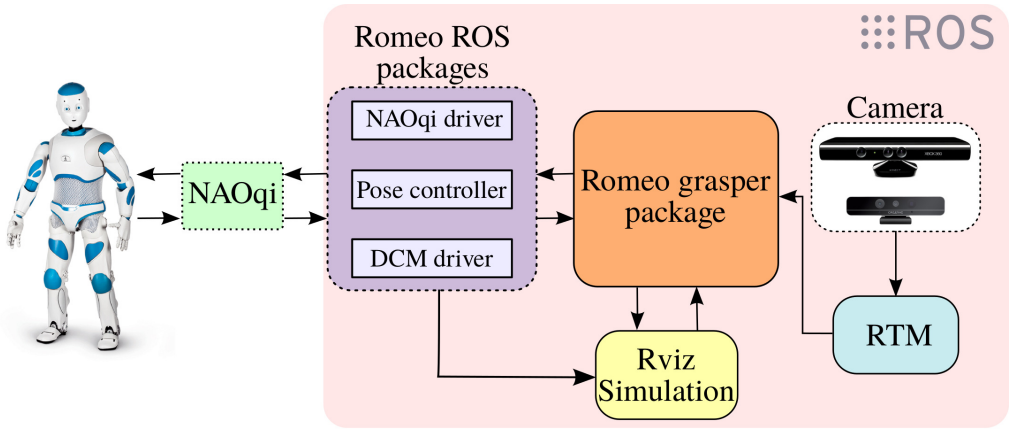


Figure 4.1: Overview of the `Romeo grasper` package

not possible to get a good performance with it because, as mentioned in Section 3.2.2, the images of Romeo depth camera are useless. Moreover, the robot is not able to see its own arms till they are too far to work with because of the range of the camera. Seeing that we need to use an external camera to get the environment information.

Although, we can use any camera that can publish `PointClouds`, explained in Section 3.2.3, is better to use the same type of camera used for the modeling of the object. In that case, we obtain a better performance of the RTM software and so less error for the global system. Once we have the camera, we can start positioning the camera.

4.1.1 Positioning the camera

In `Romeo grasper` package there are different ways to make known to the robot the position of the camera:

1. Use of pre-known position of the camera relative to any link of the robot.
2. Use a reference object that the camera recognises and associates with a link of the robot.

The first one, use the position of the link and then add the relative position to have the camera position on the robot reference and, optionally, can be set the RPY transformation for the camera. The second one uses the camera recognition software to know the position of the reference object. This object is associated with a link of the robot, so the position of the reference object is the position of the link. The ideal case would be that the reference object was itself the link robot. This case has been tested but without succeed because of the lack of `SIFT` points on the robot link surface, that makes difficult for the RTM software to detect the link. Because of that we use a reference object with an offset parameter to fix the error in the position, this offset parameter is explained in the following chapter,

Section 5.1.

Once we have the position of the camera, this is published to ROS. So it can be used for the simulator and to get the position of the object relative to the robot.

4.1.2 Calculate the object position

After having the position of the camera, we can start with the process of getting the position of the object. When the RTM software gives us the position, that is presented in camera coordinates. So we need to transform from this reference to robot reference. The way to do that is represented in the equation (4.1), so using the transform from camera reference to robot, got with the position and orientation of the camera on robot reference, and with the position on the camera reference results in the position on robot reference.

$$P_{robot}^{object} = T_{robot}^{camera} \cdot P_{camera}^{object} \tag{4.1}$$

4.2 Move the hand to object position

Once we have the position of the object on the robot reference, we need first to move the hand to a pre-grasping position to then do the picking. In order to be able of this is required an inverse kinematic solver, like the two explained in Section 3.2.4, with this we get the joint trajectory which will be followed by the arm. Although, before move it, we should see in the simulator if there is any collision or not with external objects (collisions with the robot itself is checked automatically). Because of checking this and seeing the arm trajectory, after planning, by default, the package waits the answer of the user with a service call, where you can choose: (1) to move the robot with this plan; (2) make another plan or (3) abort and wait till have another position. If we want to execute automatically the planned trajectory it only needs to put the ROS parameter `automatic_execution` to `true`. Finally, after executing the plan for the pregrasp, it starts again the function but in this case to do the picking. As explained in Section 2.5, after testing different pickings, if someone is right it will be execute.

It should be stated that as said in Section 2.2, the left arm is configured to make a side grasp and the right arm for a top grasp. However, we can decide which arm use like others wide variety of parameters, as explained below.

4.3 Configuration and launching

Firstly, describe every parameter loaded to ROS with the file `romeo_grasper.yaml`. Some of the concepts are explained in detail in the following Chapter 5

Parameter name	Default value	Description
`move_group`	left_arm	Defines which is the group of links and joints controlled, see Table 3.1 in Section 3.1.1.
`models_directory`	.../data/models	Directory containing the modeled objects and their config files, see Section 5.1
`model_object_name`	tea_box	Name of the modeled object which should be grasped.
`model_reference_name`	ref_tea_box	Name of the modeled object which is used to position the reference link, explained in Section 5.2.2
`pose_threshold`	0.05	Minimum position difference, from the old position to the current one, needed to consider that the position has changed.
`confidence_threshold`	0.25	Minimum tracker confidence to use a position. If the position does not pass the threshold the position is rejected.
`ref_confidence_threshold`	0.35	Minimum tracker confidence to use a position to set the reference link.
`reachVsGrasp`	true	Boolean to set the way to do the planning of the pre-grasp, using `reachAction` or `graspPlan`. More information in Sections 2.5 and 5.3.1
`max_velocity_scaling_factor`	0.1	Maximum factor of speed for the joints.
`tolerance_step`	0.01	Step which increments the goal tolerance for every attempt in `reachAction` and `pickAction` function.
`tolerance_min`	0.01	Initial goal tolerance for planning and picking.
`planning_time`	5	Maximum planning time for each attempt.
`attempts_max`	9	Maximum number of attempts.

continued on next page

continued from previous page		
Parameter name	**Default value**	**Description**
`base_link_frame_id`	base_link	Base frame of the robot.
`camera_in_front`	false	Boolean to know if the camera is in front of the robot or next to it. More information in Section 5.2
`camera_ref_frame_id`	base_link	Frame of the link used as reference, see Section 5.2.2.
`camera_link`	camera_link	Frame used to set the position of the camera.
`object_frame_id`	object_frame	Frame of the object that should be grasped.
`camera_pose_x`	0.0869	Coords of the camera when camera position is pre-known, more in Section 5.2.1
`camera_pose_y`	0.73	
`camera_pose_z`	-0.1897	
`camera_pose_roll`	$\frac{\pi}{12}$	Orientation of the camera in RPY when camera position is pre-known, see Section 5.2.1
`camera_pose_pitch`	0	
`camera_pose_yaw`	$-\frac{\pi}{2}$	
`camera_frame_id`	camera_rgb_optical_frame	Frame of the camera following the ROS Enhancement Proposals (REPs) as an optical camera frame [13].

Table 4.1: Description and default value of every parameter of `romeo_grasper.yaml`

Referring to the launching, in `romeo_grasper` package, there is a main launch file which is named as `romeo_grasper.launch`. This run the main node of the package loading the following parameter files:

- `romeo_grasper.yaml`: the one explained above.
- `kinematics.yaml` from `romeo_moveit_config` package: which contain the features about the kinematic solver.
- `romeo_grasper_data.yaml` from `moveit_simple_grasps`: file with the geometry of the romeo gripper.
- `model_config.yaml` for every modeled object used: configuration file of the modeled object. This file should be in the following namespace: `models/MODEL_NAME`.

Moreover, the launcher has the following arguments which are loaded as parameters for the node and these define how the package should work.

Although, a real camera were not used, so being `tracking` false, the position of the object can be given to `romeo_grasper` though the `object_tracker/object_pose` topic

Parameter name	Default value	Description
`launch_full`	false	In case of launch all the nodes together `romeo_grasper` waits some seconds to give time to the other nodes to be ready.
`verbose`	false	Enables verbose mode.
`debug`	false	Enables debug mode.
`tracking`	true	Should be enable when a real camera is used.
`pose_camera_preknown`	false	Enable use of the parameters `camera_pose_x/y/z` to set camera position.
`simulation`	false	Used when is not working with real Romeo
`automatic_execution`	false	Avoid waiting for user answer to execute the plan

Table 4.2: Description and default value of `romeo_grasper.launch` arguments.

as it is done by the RTM software.

However, the previously launcher cannot be run alone, it needs other nodes to work properly. Therefore, there are two more launchers which run these nodes automatically:

- `romeo_grasper_full.launch` to run on the real robot.
- `romeo_grasper_demo_full.launch` to run on a simulation or on the real robot but without moving it.

Both launchers run `romeo_grasper.launch`, the `romeo_full_py.launch` launcher which is the one for `naoqi_driver_py` and `romeo_sensor_py` packages and an Rviz launcher. However, the last one for the demo launch is run with `fake_execution`, so it doesn't move the robot. Furthermore, they need a way to move the robot, when is with the real robot is though `DCM` package and with the simulation is though `naoqi_pose` package. Therefore, the real launcher runs the `DCM` driver and the demo launcher runs the `pose_controller` node. For more information about these last packages and nodes see Section 3.2.2.

It needs to be stated that to use a real camera is necessary to launch another node. There are two possible cases: (1) use a RealSense camera or (2) use a camera through openNi.[1] On one hand, the RealSense camera can be launch with the `use_realsense` argument in the launchers. On the other hand, for the openNi case, should be run the following command on a new terminal:

```
roslaunch openni_launch openni.launch depth_registration:=true
```

[1]Installation of the camera drivers explained in Appendix A.3.

Finally, it should be mentioned that the Rviz configuration and launching files are copied from `romeo_moveit_config`, but modified to have a more adapted features for this package.

Chapter 5

Implementation

At this stage the details of every process are explained. So the parts briefly exposed previously are analysed deeply. Sometimes are used knowledge described in previous chapters, but they have the corresponding reference to make easier the reading.

5.1 Model Object class

This class represent, in the `Romeo grasper` package, an object that has been previously modeled with the RTM-Toolbox. This class allow to create, remove and update every object independently in the simulator and to calculate its own position with the pertinent correction on the robot reference, given the position on camera reference.

At the initialization of this class, it should be associated with one of the modeled object with the pertinent configuration file `model_config.yaml` in the same folder of the modeled object.[1] The config file has the following parameters:

- **offset_x/y/z**: the origin established by the RTM-Toolbox may be is not the desired one for the grasp. This offset use the orientation provided by RTM-Toolbox.
- **shapeType**: shape used in the simulator. For now only supports `cylinder`.
- **size**: set the radius of the cylinder.
- **size_l**: set the longitude of the cylinder.

[1] In the package the folder for every model is placed at `data/models/MODEL_NAME`.

Every modeled object that wants to be used must have the configuration file loaded in the launch file. So the following line should be used for every model in the node namespace *MODEL_NAME* for the name of the model.

```
<rosparam command="load" ns="models/MODEL_NAME" file="$(find romeo_grasper)
    ↪ /data/models/MODEL_NAME/model_config.yaml" />
```

An overview of the processes done by this class is showed below, on Figure 5.1. Moreover, the last part of the flow sequence is explained in detail on Figure 5.2.

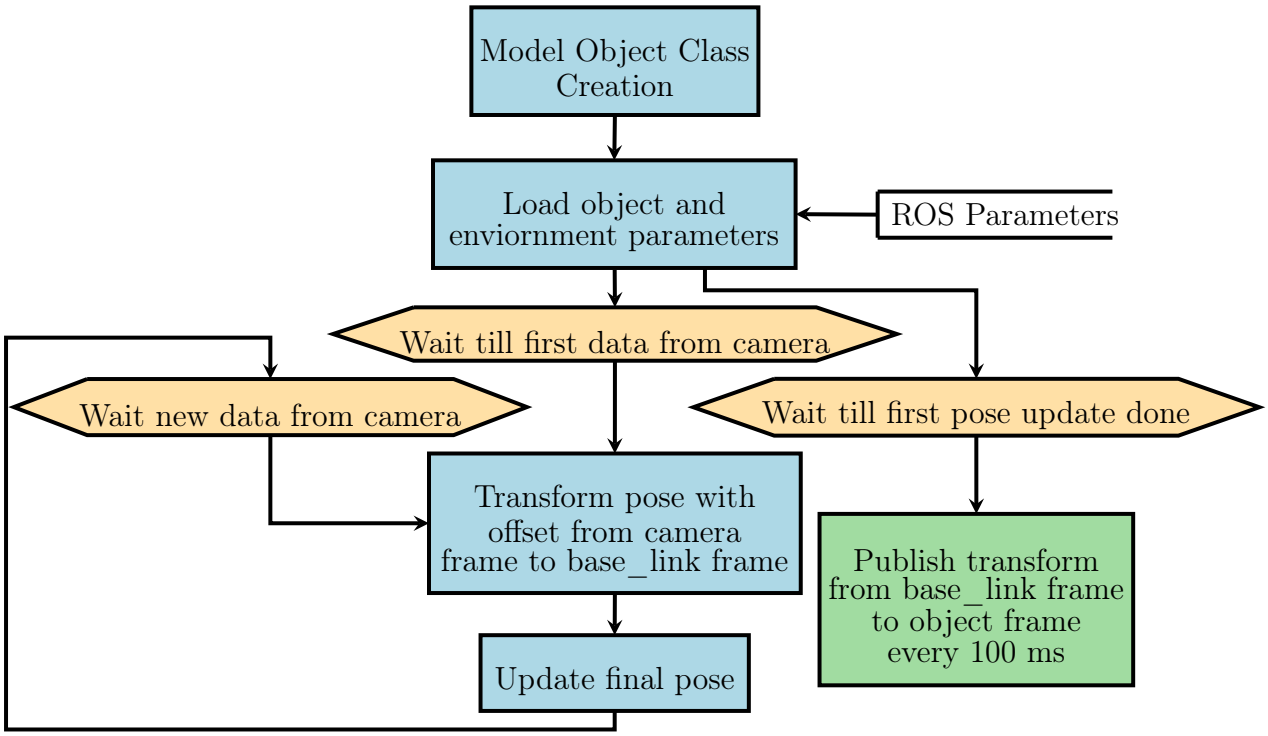


Figure 5.1: Flow sequence of ModelObject class

5.2 Approach to camera position

The aim of having the position and orientation of the camera is to have a transformation from camera reference to base reference. So when the position of the object is given on camera reference, we can transform this to base reference, and then do the grasp.

Previously in Section 4.1.1, has been explained a summary of how to let known to the robot the camera position. As mentioned previously, we have to ways of doing that: (1) using pre-known distance between a link of the robot and the camera; (2) using the camera to know its own position. These two ways are described in detail in Section 5.2.1 and 5.2.2, respectively.

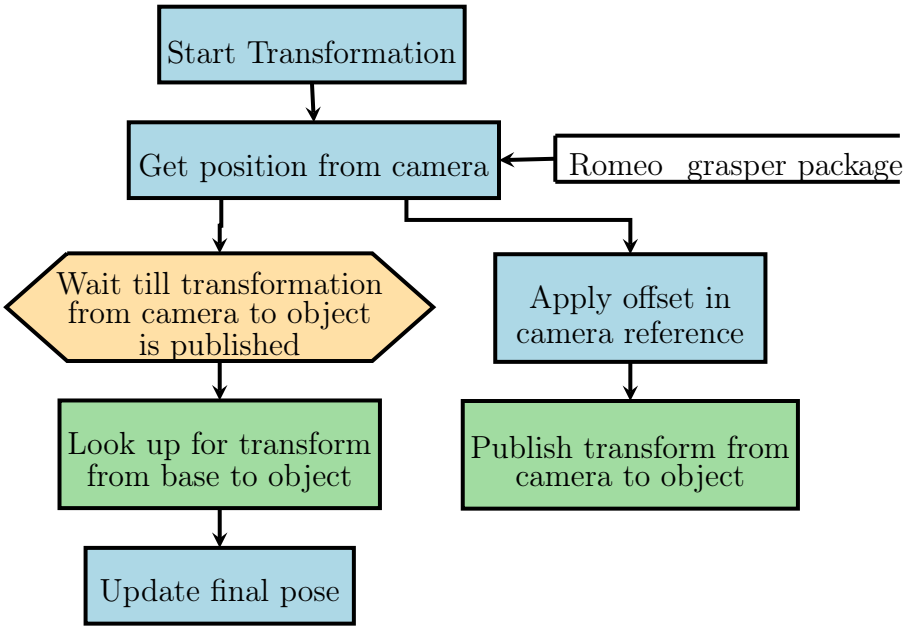


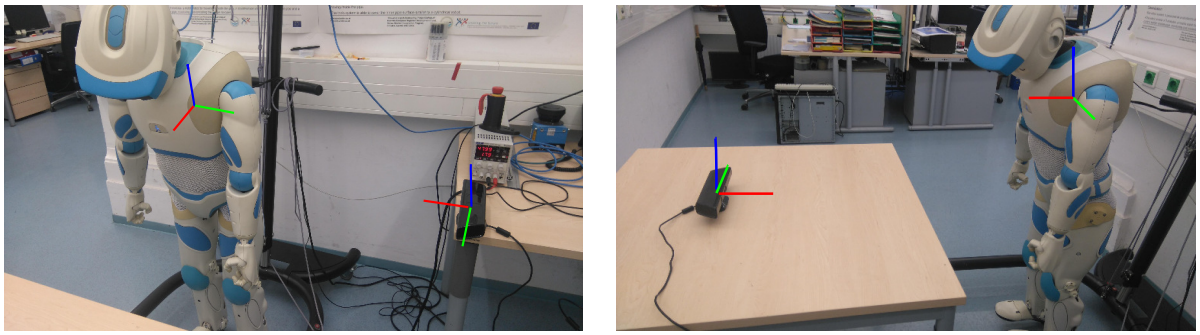
Figure 5.2: Position transformation from camera reference to base reference

Firstly, these two ways start with the assumption that we have the camera perfectly orientated at 90 degrees or 180 degrees in yaw rotation. So the camera is next to the robot or in front of the robot, as in the Figures 5.3a and 5.3b, respectively. Therefore, the matrix rotation from `camera_link` to `base_link` for this cases are expressed as Equation (5.1) for camera next to the robot and as Equation (5.2) for in front. Then, when we configure the package, with file `romeo_grasper.yaml` we should indicate which case is the current orientation by the boolean parameter `camera_in_front`. So this estimated orientation is used, depending of the way to get the position, as the final orientation or only as a seed.

$$S_{base}^{cam} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix} \tag{5.1}$$

$$S_{base}^{cam} = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix} \tag{5.2}$$

It should be stated that to let know the camera position to the simulator and others part of the package is used a separated thread. This one is in charge of publishing the position of the camera on `TF` when this is available. Moreover, if currently is not using a real camera it publishes the pose of the camera optical frame according to REP 103 [13], which is used to get the position of the object, as explained in 4.1.2.



(a) Next to the robot (b) In front of the robot

Figure 5.3: Orientations of the `camera_link` respect to the robot `base_link`

5.2.1 Use of pre-known distance

In an environment where we can know the position of the camera respect to a link of the robot, use this way is a good option. Therefore we only need to set the configuration in the `romeo_grasper.yaml` file. So with the `camera_ref_frame_id` parameter we set the link used as reference and with `camera_pose_x/y/z` specify the coordinates where is the camera. However, we have to take in account that the orientation for this parameters is the `base_link` orientation. That is done in that way to facilitate the user to take the coordinates.

Therefore, the process to calculate the position of the camera is, as described in Equation (5.3), taking position of the reference link in `base_link` coordinates and adding the values of the parameters `camera_pose_x/y/z`.

$$P_{base}^{cam} = P_{base}^{link} + \begin{pmatrix} camera\_pose\_x \\ camera\_pose\_y \\ camera\_pose\_z \end{pmatrix} = P_{base}^{link} + \left(P_{base}^{cam} - P_{base}^{link}\right) \tag{5.3}$$

Moreover, optionally we can set the orientation transformation from `base_link` to `camera_link` in RPY using the parameters `camera_pose_roll/pitch/yaw`. Therefore the S_{base}^{cam} matrix is calculated with these parameters or, in case they don't exist, use the initial assumption. Finally, the transformation between `camera_link` and `base_link` is expressed like Equation (5.4)

$$T_{base}^{cam} = \left(\begin{array}{ccc|c} & S_{base}^{cam} & & P_{base}^{cam} \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{5.4}$$

An overview of the flow sequence done by the package in order to get this transformation is described in Figure 5.4.

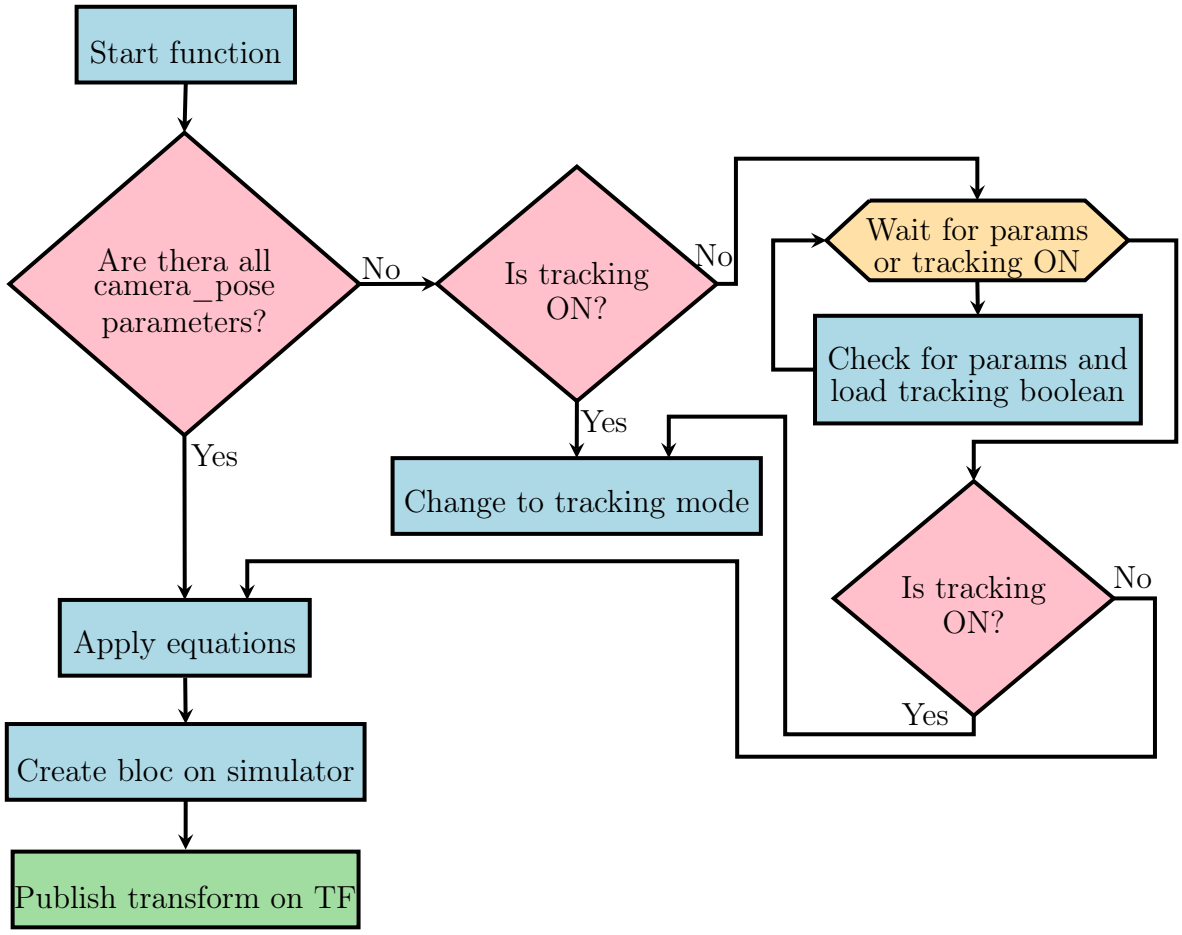


Figure 5.4: Process of having camera position using pre-known distance

5.2.2 Use of visual recognition

The original idea of this method is that wouldn't be necessary the help of the user to get the camera position. So the package should recognise one or more parts of the robot. Then, it should calculate the position and orientation of the camera knowing the position and orientation[2] from the `camera_link` reference and from the `base_link` reference. For now, this initial idea is not totally implemented due to difficulties to recognise parts of the robot, as mentioned in Section 4.1.1, and for others problems explained later.

In order to be able to get the pose of the links of the robot is used a modeled object with a special offset and easy to recognised for the software. This object is placed next to the link of the robot in a precise location, as showed on Figure 5.5, so the final origin

[2]The use of the orientation of a robot part depends on the approach used.

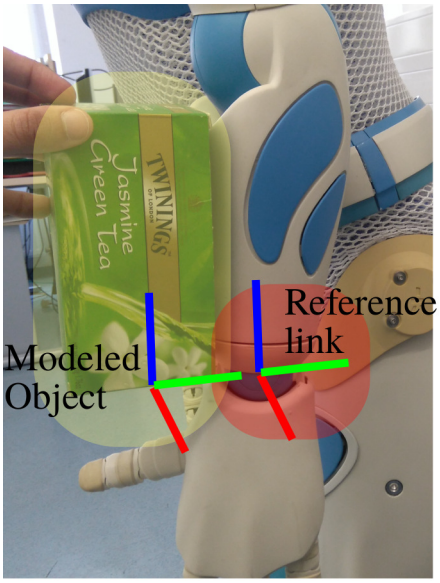


Figure 5.5: Position change due to the offset of the modeled object.

of the object is established on the link origin. This modeled object, when it's initialised must have boolean `is_reference_object` with value `true` to work properly.

Below are explained the three different approaches done for this method. Although, only the first one is implemented in the package, the others are prepared in the Matlab files on the directory `data/matlab/`.

(1) Using only one link

In this case, is used the assumption of having the camera next to or in front of the robot as a final orientation. Therefore, the only unknown variable is the camera position on base reference, P_{base}^{cam}. So as to solve this is needed the position of a link on camera reference, P_{cam}^{link}, that is known also on base reference, P_{base}^{link}. Finally the Equation (5.5) is the one to solve.

$$P_{base}^{cam} - P_{base}^{link} + S_{base}^{cam} \cdot P_{cam}^{link} = 0 \tag{5.5}$$

(2) Using three links

On the other hand, this case only used the assumption of the orientation to have a seed to solve the problem. The idea of this way is to implement the resolution of an optimisation problem with non-linear constraints to have an exact position and orientation of the camera.

The equation that have to be minimized are Equation (5.5), for every link, and Equation (5.6), for two combination of the three links.[3] Therefore, the error of every equation is put as absolute value and introduce in the minimization function.

Moreover, the solution have three non-linear constraints: the three axis of the orientation should be perpendicular between them. The axis of the orientation are the unitary vectors of the columns of matrix S_{base}^{cam}. Therefore, the constraints are Equation (5.7), for every combination of the axis.

$$\left(\overrightarrow{P_iP_j}\right)_{base} - S_{base}^{cam}\left(\overrightarrow{P_iP_j}\right)_{cam} = 0 \qquad \forall i \neq j \qquad i,j = [link_1, link_2, link_3] \tag{5.6}$$

$$< v_i, v_j >= 0 \qquad \forall i \neq j \qquad i,j = [x,y,z] \tag{5.7}$$

The reason to use three link for this case is that a minimum of 12 equation are required to determine the 12 variables (3 for the camera position and 9 for the orientation). Using three links we get 15 equations and 3 non-linear constraints. Otherwise with two link we would have 9 equation and 3 non-linear constraints, which is not enough.

As said previously, the fact of having the camera in front or next to the robot is used as a seed. So the depending of the case is used the matrix from Equation (5.1) or from Equation (5.2). Furthermore, for the position there are also seeds, these are extracted experimentally.

Finally mention that in Matlab is used the optimization solver `GlobalSearch` because it allows non-linear constraints and also search in a big range of solutions. Therefore, apart from looking for the best solution from the given seed, it generates some trial points in all the space inside the constraints. These trial points are potential start points as the seed, so this method allow to not remain in a local minimum.

(3) Using multiples transformations

The RTM software, apart from giving the position of the object, gives the orientation. So instead of only using the position of the link placing the reference object at a certain place, we could use also the orientation of the object. Therefore, we could discover the position and orientation of the camera placing the object in a precise position and orientation from the known link. In order to do that we should know: (1) the transform

[3] Only two combination of Equation (5.6) because the third one is linear dependent of the other two.

from object to link reference, because of that reference object is placed by us; (2) the transform from link to base, provided by the model of the robot; (3) the transform from camera to object, known by using the information of the RTM software. With all this information we could have the transformation from camera to base, as showed in Equation (5.8), so finally get position and orientation of the camera.

$$T_{base}^{cam} = T_{base}^{link} \cdot T_{link}^{obj} \cdot T_{obj}^{cam} \tag{5.8}$$

5.3 Planning and execution of trajectory

A brief description of this part is in Section 4.2. Firstly, before planning the ROS parameters named as runtime parameters are constantly updated. And secondly, is needed to accomplish the following conditions:

- The modeled object which should be grasp has the position updated.
- The position of the object is a new one which has not been processed.
- Be sure is not doing other processes that can effect the planning and execution

After that, is time to start with the planning and execute function, an overview can be seen in Figure 5.6. Therefore, below is explained in more detail some parts of this function.

5.3.1 Planning pre-grasp

As said in Section 2.5, there are two ways of planning the grasp. Therefore, here depending on the parameter `reachVsGrasp` can be chosen which way.

On one hand, for the `reachAction` case, first is set the flag to not move the arm and then is called the `reachPregrasp` function. This function call `reachAction` but modifying the target pose subtracting the coords from the gripper, stored in the file `romeo_grasp_data.yaml` of package `moveit_simple_grasps`. Therefore, it goes to a position near the object but without touching it.

On the other hand, for the `graspPlan` case, using the block from the modeled object generates a large number of grasps and from them takes the variable `grasp_pose` which defines the position of the parent link of the end effector for the grasp. All these positions, from every grasp, are given to the `Action` class as target positions. Therefore, to make the plan will test them and will choose one which doesn't collide.

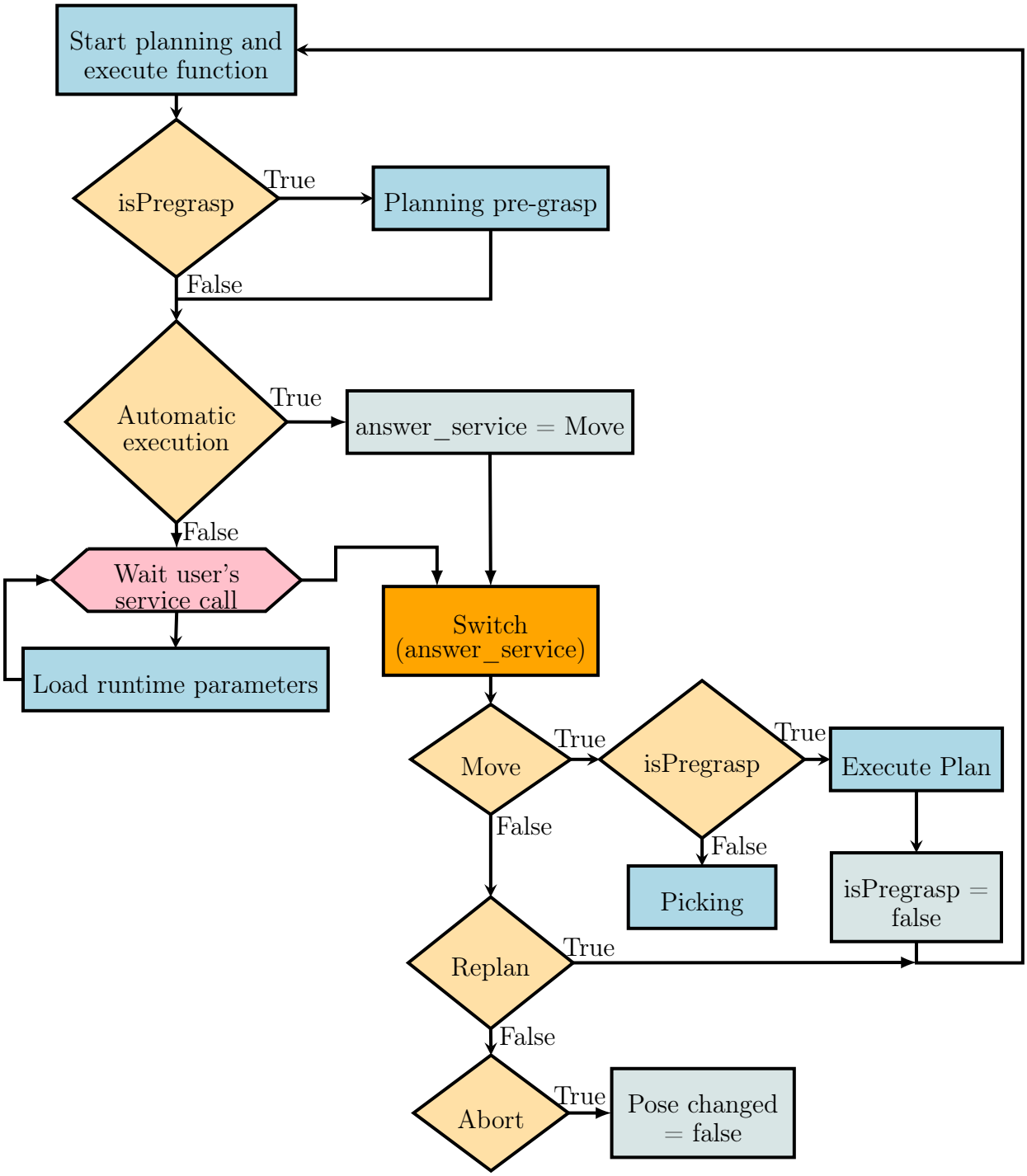


Figure 5.6: Flow sequence for function responsible of the robot planning and execution

Finally, both ways ask to the `move_group` class for a plan. This plan contain the information of: (1) the starting state of the robot used for planning; (2) the planned trajectory for the joints of the group and (3) the amount of time it took to generate the plan.

5.3.2 User answer service

Before executing the movement there is a loop waiting for the user interaction with the package. This interaction is accomplish by a service call requesting for: (1) move the arm, (2) make the plan again, (3) abort plan. Therefore, this allows the user to see the planning trajectory before executing the movement. Moreover, the pose of the object is updated in another thread and while the loop is running it loads constantly the runtime parameters. As a result of this, the parameters or goal can be modify and then make the plan again. However, there is also the option of don't have this loop and execute directly the plan putting the parameter `automatic_execution` to true.

The implementation in `romeo_grasper` of this loop is waiting till one of the three following services is called:

1. `romes_grasper/abort_plan` change the value of the variable `pose_changed` to false. Therefore, the planning and execute function won't start till the object position has changed again.

2. `romes_grasper/execute_plan` starts with the movement of the arm to go to the pre-grasp position or to do the picking. Then, it also change the value of the variable `pose_changed` to false.

3. `romes_grasper/replan` starts again the planning and execute function

5.3.3 Execute pre-grasp plan and picking

As mentioned in Section 4.3, `romeo_grasper` can be launched with simulation or with real robot. Therefore, depending on it the execution of the pre-grasp is started: (1) through the Action class of `romeo_moveit_action` package when is to the real robot or (2) through the `RomeoSimulator` class, explained in Section 5.3.4, when is to a simulation.

On one hand, when the `executeAction` function from Action class is called, it runs the `execute` function of `move_group` class with the previously created plan as a parameter. Then the MoveIt class made contact with the Controller Manager to move the robot according to the trajectory in realtime and, as explained in Section 3.2.2, it works with the DCM driver to send the proper commands to the robot. On the other hand, the execution with `RomeoSimulator` class is launched by the function `executeTrajectory`. This function and how `RomeoSimulator` class works is explained in below. Finally, the picking is launched by the `pickAction` function of the Action class, the way how it works is described in Section 2.5.

5.3.4 RomeoSimulator class

The aim of `RomeoSimulator` class is a way to move the simulated Romeo on your computer as the trajectory previously created. In order to do this, it is subscribed to the topic `/trajectory`, where the Action class publish the trajectory, see Section 2.5. Then, every time a new trajectory is published, `RomeoSimulator` update its internal trajectory variable. Therefore, when the `executeTrajectory` function is called, the class can use the trajectory with the most recent version. Finally, the `executeTrajectory` call the action service `/joint_trajectory` from `pose_controller` node, explained in Section 3.2.2.

5.4 Setup of romeo grasper

Once the main idea of the `romeo_grasper` package is described, it should be explained the required setup, an overview is shown in Figure 5.7. Firstly, the services for user choice after planning and the parameters detailed in Section 4.3 are loaded. Then the `RomeoSimulator` class is initialized. After this, two new thread are created to make the process faster. Therefore at this stage there are three threads at the same time:

1. Initialize the visual environment, spawn a table as a cuboid and create a subscriber to `/romeo_grasper/visual_table` which is used to modify the features of the table.
2. Create the MoveIt Actions for every arm and set the current parameters to these.
3. Create the subscriber to `/object_tracker/object_pose` to know where is the object and in case of `tracking` initialize the client service to change tracking model.

Finally, it waits till all the setup processes are done and then starts getting the camera position which is summarized in Section 4.1.1 and explained deeply in Section 5.2.

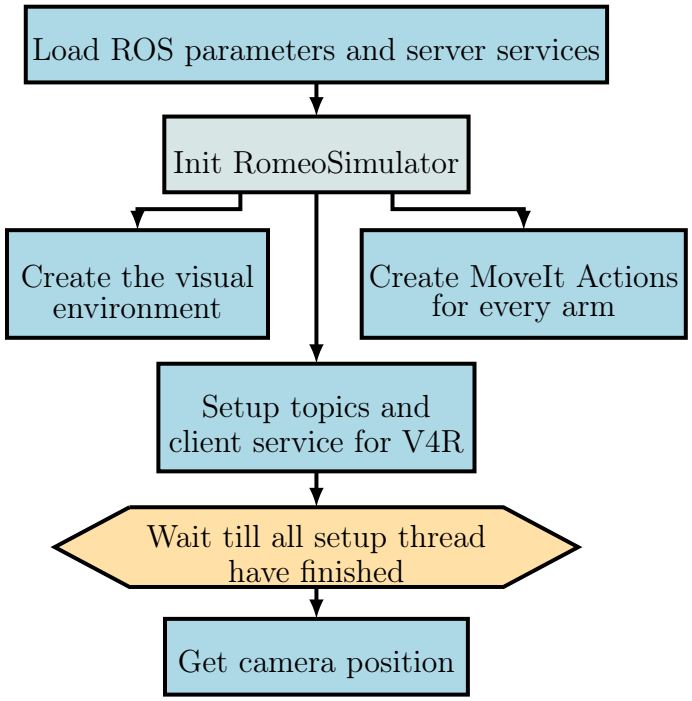


Figure 5.7: Flow sequence for `romeo_grasper` setup

5.5 Communication between packages

Romeo grasper depends on other packages to work properly, so it needs to sent and receive data to make Romeo achieve the grasp of the object. An overview of the communications involve in this process is shown in Figure 5.8. Furthermore, the explanation of every arrow of the Figure 5.8 is described in the following Table 5.1.

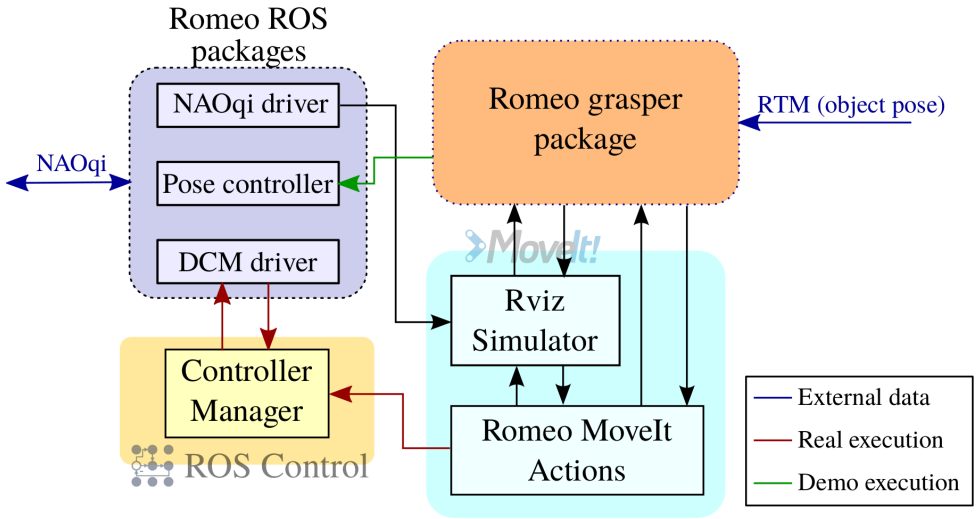


Figure 5.8: Flow of data from object pose to commands to NAOqi

Sender	Receiver	Data	Description
RTM	Romeo grasper	Object pose	Information of the object position and the confidence of the tracker
Romeo grasper	MoveIt Actions	Commands	Commands to plan, execute, pick or set parameters to the `move_group`.
MoveIt Actions	Romeo grasper	Results	Returns the results of the commands sent, like if it is succeeded, the trajectory planned or the final planned pose.
Rviz Simulator	Romeo grasper	Link poses	Position of some elements like the wrist or the camera are sent to calculate the camera pose or the object pose.
Romeo grasper	Rviz Simulator	Link poses	Position of the camera and of the object.
NAOqi driver	Rviz Simulator	Joint state	Simulator needs the joint state to have the simulated robot as the real one.
MoveIt Actions	Rviz Simulator	Commands	Commands to calculate a plan and simulate to check collisions.
Rviz Simulator	MoveIt Actions	Results	Returns if it is succeeded or the trajectory planned.
MoveIt Actions	Controller Manager	Execution Plan	Sends the plan which should be execute on the real robot.
Controller Manager	DCM driver	Commands	Commands with the right timing to have a real-time control with the Romeo robot.
DCM driver	Controller Manager	Joint state	Controller Manager should be update constantly with the Romeo state in order to be able of having a real-time control.
Romeo grasper	Pose controller	Trajectory	In case of simulated robot, it sends the trajectory to be executed.
Romeo ROS	NAOqi	Commands	Commands to move the robot and to get specific information.
NAOqi	Romeo ROS	Data	All kind of data: joint state, camera images, variable names, etc.

Table 5.1: Information sent and received from and to the packages.

Chapter 6

Experiments

6.1 Movement in simulation

On the following experiment has been tested how accurate is the movement in the simulation to go to the goal position. In order to do that has been created a program to automatically introduce new positions and catch the results of the process. The program can be found in the folder `src` with name `experiment1.py`. Unfortunately, not all the used information has been caught by the program and has been manually introduced looking to the terminal log. Moreover, initially the program didn't work well and a lot of tests doesn't have the data to confirm the observations made. Therefore, some information is given from the perception of the author of the dissertation. The main results used for this experiments are: (1) the success of the movement; (2) the distance from the goal position to the position where the trajectory calculated by the solver moves the hand; and (3) the planning time spent.

Whereas the tests has been made all the time with the same goal positions, shown in Table 6.1. The goal is a cylinder with radius 3.5 cm and longitude of 15 cm put it in both orientation vertical and horizontal. Furthermore, it has been run on two different computers (`C0` and `C1`). On one hand, the computer named `C0` has a better graphic card the `C1` has a better processor. About the software it has been tried to execute the same on both computers.

As explained in Section 2.5, there are two ways of calculate the trajectory for the grasp: (1) with `reachAction` function and (2) with `graspPlan` function.[1] Therefore, in this experiment it is tested both ways to see their particularities. Moreover, as also

[1] To test graspPlan it is needed to put to zero the offsets of the `romeo_grasp_data.yaml` file from the `moveit_simple_grasps`. Otherwise it cannot know the error in the trajectory.

	Position [m]		
	x	y	z
Cylinder 1	-0.3	0	0.5
Cylinder 2	-0.25	0	0.45
Cylinder 3	-0.20	0	0.40
Cylinder 4	-0.15	0	0.35
Cylinder 5	-0.05	0	0.35
Cylinder 6	-0.3	-0.1	0.5
Cylinder 7	-0.25	-0.1	0.45
Cylinder 8	-0.20	-0.1	0.40
Cylinder 9	-0.15	-0.1	0.35
Cylinder 10	-0.05	-0.1	0.35

Table 6.1: Positions in camera reference of the cylinders used for the tests.

described in Section 2.5, the `reachAction` function has two methods of solve the inverse kinematics: (1) looking for solutions inside a given goal tolerance and (2) solving with approximated solutions without taking in account the goal tolerance.

Besides, one of the other important factors is the maximum time for the planning. Therefore, it is tested different possibilities of planning time.

Then, for this experiment has been made 178 different tests combining, `graspPlan`, `reachAction` (toleranced or approximated solutions), different planning time and with both computers. Even more which has not been introduced in the statistics but taken in account by me. Some observations of the results are the following ones:

- 32 solutions inside the tolerance found out of 77. The rest are solved with approximated solutions with computer `C1`.
  - 10 out of 20 with 2 seconds of planning time.
  - 12 out of 37 with 5 seconds of planning time.
  - 10 out of 20 with 60 seconds of planning time.
- 6 solutions inside the tolerance found out of 60. The rest are solved with approximated solutions with computer `C0`.
  - 2 out of 20 with 2 seconds of planning time.
  - 2 out of 20 with 5 seconds of planning time.
  - 2 out of 20 with 20 seconds of planning time.
- Average distance of `graspPlan` is 0.0916 m.

  - 4 out of 14 solutions with `graspPlan` found with 2 seconds of planning time by computer `C1`. These solutions have a mean of 0.077 m.
  - 6 out of 7 solutions with `graspPlan` found with 60 seconds of planning time by computer `C1`.These solutions have a mean of 0.136 m.
  - 2 out of 16 solutions with `graspPlan` found with 2 seconds of planning time by computer `C0`.These solutions have a mean of 0.071 m.
  - 10 out of 21 solutions with `graspPlan` found with 20 seconds of planning time by computer `C0`.These solutions have a mean of 0.075 m.

- The `graspPlan` function after planning spend a lot of time doing something unknown (there is not any log in the terminal). Therefore, the real planning time is different and depends on the computer. Moreover, there is no differences changing the planning time.
  - Average real planning time of computer `C1` is 250 seconds.
  - Average real planning time of computer `C0` is 135 seconds.
- Average times:
  - Approximated solutions always last less than 1 second, usually with a mean around 0.1 seconds.
  - Solutions inside tolerance are usually below 5 seconds, with a mean around 1.5 seconds and it has been two isolated cases of 20 seconds out of around 40 samples.
  - `graspPlan` solutions with 20 seconds of planning time have a mean of 8.35 seconds and the ones with 60 seconds of planning time have a mean of 10.39 seconds.
- It has been noticed it gets more solutions inside the goal tolerance if instead of large planning times uses small planning times but with a lot of attempts. The best way is starting in a low goal tolerance, using a low goal tolerance step and do a lot of attempts increasing the goal tolerance at each attempt.

After that, it should be shown a comparative of the resulting distance with the approximated solution between the two computers and with planning times of 2 and 5 seconds. That can be seen in Figure 6.1.

Finally, for this experiment has been used a maximum tolerance of 0.10 m. Below, the confidence interval of the distance for approximated solutions and for solutions inside

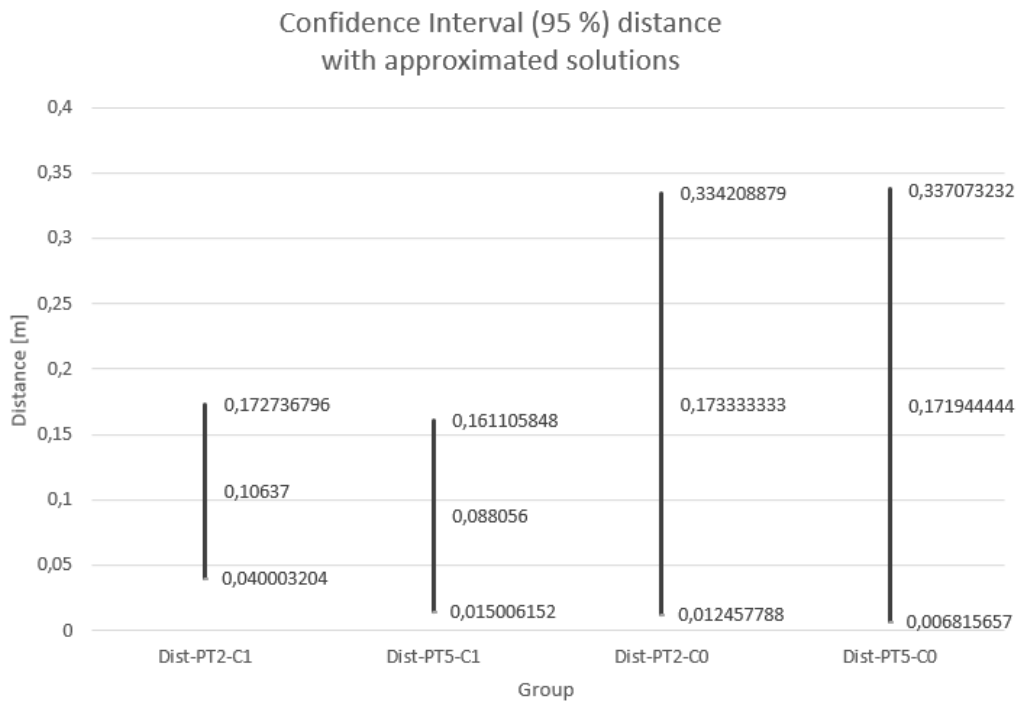


Figure 6.1: Confidence interval of distance of the approximated solutions.

tolerance are shown in Figure 6.2.

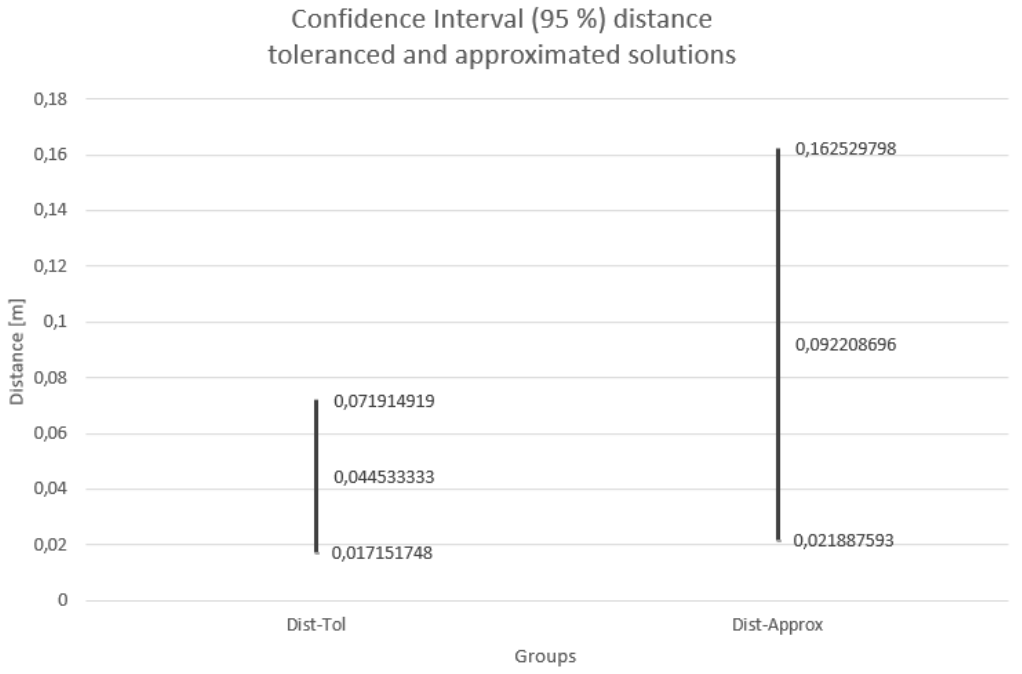


Figure 6.2: Confidence interval of distance for approximated and inside tolerance solutions. Only for solutions with computer `C1`

6.2 Implementation of IKFast on Romeo

Currently, the inverse kinematic solver used is from the KDL library, explained in Section 3.2.4. However, it gives some results that could be better and the planning time is an important factor as shown above. Therefore, a solution for this problem could be implement the IKFast solver, also explained in Section 3.2.4, which is more precise and faster, usually it last around 4 ms to calculate the operations [12].

In order to implement the IKFast on the Romeo robot, firstly is needed to generate the C++ code which solve the inverse kinematics. However, This OpenRAVE software has several dependencies to generate the C++ code, explained with the installation in Section A.5. Secondly, once the code is already generated, it should be create a package which provides the kinematic solver, based on the previous C++ code. Although, it is needed to have the `moveit_ikfast` ROS package to create the kinematic solver package.

Referring to the code, it is created from a Collada model file which is not found in the `romeo_description` package. Then, this can be created, using the command below, from an URDF model of Romeo which is contained in the previous package.

```
rosrun collada_urdf urdf_to_collada romeo.urdf romeo_raw.dae
```

After that, as recommended by the author of IKFast [12], it is good to round at the fifth decimal the values of the Collada model to make easier to find aligned axes. The rounding program is found in the `moveit_ikfast` package, more information in [21], and is run with the following command:

```
rosrun moveit_ikfast round_collada_numbers.py <myrobot_name>.dae <
    ↪ myrobot_name>.rounded.dae 5
```

Once the Collada model is ready, it is time to generate the IKFast solver code. As said in Section 3.1.1, the Romeo arms have 7 DoF, so the inverse kinematic type should be `transform6d` with one redundancy to be able to go to a specific position and orientation. Moreover, has to be defined which is the base link and the end effector of the group. So between these are contained the joints of the arm, Table 3.1 can help to know parent and child link for every joint. The index of the arm links of Romeo are shown in Table 6.2.

Therefore, the generation process is executed with the command below. Finally, it should be stated that if for the configuration are contained 7 joints, it should be add the `freeindex` parameter defining which is a free joint.[2] The index for the joints is shown in Table 6.3.

[2]The free joints are specified before the IK is run, these values are known at runtime, but not known at IK generation time [12].

Body	
Link	**index**
base_link	0
torso	1

Left arm		Rigth arm	
Link	**index**	**index**	**Link**
LShoulder	2	43	RShoulder
LShoulderYaw_link	3	44	RShoulderYaw_link
LForeArm	4	45	RForeArm
LElbow	5	46	RElbow
LWristRoll_link	6	47	RWristRoll_link
LWristYaw_link	7	48	RWristYaw_link
l_wrist	8	49	r_wrist
l_gripper	18	59	r_gripper

Table 6.2: Index of every link of the robot base and arms.

```
python `openrave-config --python-dir`/openravepy/_openravepy_/ikfast.py --
    ↪ robot=<myrobot_name>.rounded.dae --iktype=transform6d --baselink=<
    ↪ base_index> --eelink=<end_effector_index> --savefile=<
    ↪ ikfast_output_path>
```

Left arm		Rigth arm	
Joint	**index**	**index**	**Joint**
LShoulderPitch	0	16	RShoulderPitch
LShoulderYaw	1	17	RShoulderYaw
LElbowRoll	2	18	RElbowRoll
LElbowYaw	3	19	RElbowYaw
LWristRoll	4	20	RWristRoll
LWristYaw	5	21	RWristYaw
LWristPitch	6	22	RWristPitch
LHand	7	23	RHand

Table 6.3: Index of every joint of the robot arms.

After that, it is time to generate the IK solver code. As explained in Section 3.1.1, the usually way of working is to split the arm in two group. So the right way is to take the first 6 joint of the arm in the main group which is the one controlling the position and orientation. Therefore, the logical configuration is with `-eelink` equal to 7 or 48 depending on the left or right arm and `-baselink` equal to 0 or 1.[3]

Unfortunately any of this configuration can generate a C++ code with the Romeo model. The problem is that, after one or two hours of execution of the command above,

[3]It doesn't matter if is used `base_link` or `torso` link because they are fixed, so there is only a fixed transformation between them.

the program, and sometime also the computer, is frozen. Then, it has to be tested others configuration which are less logical but also useful. Therefore, these are taking the 7 joints but adding the `freeindex` parameter. The most logical free joint is the `WristRoll` because is the one closest to the end effector, doing the same movement than `ElbowRoll` and it is the slowest joint of the arm. Finally, neither this configuration nor any other joint as the free joint can generate the IK solver code.

Currently, there is a commit in the Natalia Lyubova repository[4] where the URDF model has been changed to use a new convention of the joints. But, it also gives the same results.

It should be stated that to be sure the installation it was well done, it has been tried to generate the IKFast for another robot and it worked. Moreover, there is a discussion in the official repository of Romeo package where all this IKFast issue is discussed [2].

6.3 Get pre-known camera position

Another very important factor which can lead to have some deviation on the results is to get the right position of the camera. As said before, there are two ways to get it: (1) with a pre-known position, which is the one tested here; and (2) from the visual information got from the camera itself, tested in the next Section 6.4. So below is explained how is got the position of the camera respect to the robot empirically. However, it is only tested for the camera on the left side.

Currently, the experiments are made with the Romeo robot secured with a support holding it without touching the floor. However, this makes that the robot is tilt forward. Therefore, the X and Z coordinates are difficult to measure due to the Romeo tilt. Moreover, the `base_link` origin is inside the robot, so it is difficult to get where is it precisely. So the solution to be able to get a good estimation of the position is using a Romeo link as the `l_wrist` which its origin is easy to find. Furthermore, this link can be moved easily in a position aligned with the Y axe to the camera, so the X and Z coordinates of the `l_wrist`, which are given by the simulator in the `base_link` reference, are the same as the camera. Then, the only needed measure is in the Y axe where the Y camera coordinate is the Y `l_wrist` coordinate plus the offset from `l_wrist` to the camera. However, it should be stated that this measure can have some error, which is estimated to be around $\pm$ 1.5 cm.

Therefore, the experiment is done as explained above and result with the following Table 6.5:

[4] `https://github.com/nlyubova/romeo_robot/commit/05bf217b73c389c461414412fd43d84ecb0c4115`

	`l_wrist` position	Offset	Camera position
x	0.0869 m	0.0 m	0.0869 m
y	0.470 m	0.26 m	0.730 m
z	-0.1897 m	0.0 m	-0.1897 m

Table 6.4: Measures in `base_link` reference to get camera position.

After that, the next information to get is the orientation of the camera. As the robot is tilt forward and the camera is on next to the robot aligned with the Y axe of the robot, can be estimated that the Z axe of the optical camera frame is the Y robot axe, but with the sign changed.[5]

As the origin of the robot is fixed in the simulation, on the camera is where it has to be applied all the rotation. Then, it is needed to measure the tilt angle, with the experiments it has been worked with an approximate angle of 15º. So the final transformation of the camera is the one shown in Table 6.5.

	Position [m]			Orientation [rad]		
	x	y	z	roll	pitch	yaw
Camera from `base_link` frame	0.0869	0.730	-0.1897	$\frac{\pi}{2}$	0	$\frac{11\pi}{12}$

Table 6.5: Transformation from `base_link` frame to camera.

6.4 Visual camera positioning

The method presented above is used to get the position of the camera precisely, but repeat this process every time that the robot or the camera has been moved may be is not the most comfortable way. Therefore, the process automation is the next step. In Section 5.2.2, is shown the three different possibilities to get the position of the camera using the visual information of the camera itself. Below this three ways are tested to see how good precise are they. Therefore, they have to be compared with the most accurate information which is the one got in the previous experiment. It should be stated that place the modeled object to get the position of the links has an error estimated around ± 2 cm. However, the most precisely estimated link position is for the `l_wrist`, then the `LElbow` due to use the same offset for both but it is optimised for the `l_wrist`, and the `LShoulder` can have less precision due to it was to apply an extra offset because the camera didn't recognise the modeled object in the right place.

[5] Acoording to REP 103 [13], the convention of the optical frames of the cameras are: Z forward, X right and Y down.

(1) Using only one link

The first approach is based on the information of only one link and using the assumption of being perfectly oriented in front or on side of the robot. Therefore, given the position from camera of the link and the position of the link in robot reference (information extracted from the simulator), applying Equation (5.5) results on the position of the camera. Moreover, for the next approach it is needed this information also for other links, so it can be used to see if the final result is changing between different data. Therefore, the following Table 6.6 shows the information every link with the result with that information:

Link used	Coord	Position [m] from		Camera position [m]
		camera	`base_link`	
`l_wrist`	x	-0.1297	0.1974	0.0677
	y	0.1588	0.1763	0.7294
	z	0.5081	-0.2990	-0.1401
`LElbow`	x	-0.1328	0.1180	-0.0148
	y	-0.0116	0.1858	0.7387
	z	0.5079	-0.1259	-0.1375
`LShoulder`	x	-0.1386	0.0502	-0.0884
	y	-0.2637	0.1899	0.7862
	z	0.5513	0.0739	-0.1898

Table 6.6: Data and results for the first approach with visual positioning.

(2) Using three links

The next experiment is on the second approach solves a non-linear problem which use the information of three links and then, given a seed, tries to minimise Equation (5.5), for every link, and Equation (5.6), using as constraints Equation (5.7). The data used for this is the same as used above, and to calculate the solution it has been done with `Matlab`. Moreover, the Matlab code of this can be found in the `romeo_grasper` repository, specified in Sectio 1.4, in the folder `/data/matlab`.

In order to make the experiment has been given three types of seeds: (1) with the exact position and orientation of the camera; (2) without the exact orientation, given as it is in the orientation assumption and (3) as before and also an imprecise position. Therefore, the seeds are as in the following Table 6.7:

Applying the positions of the links of the Table 6.6 and with the different seeds the final results are the ones in Table 6.8. Moreover, to approach the position of the camera has been tested to average the results of the Equation (6.1), with all the link, giving as a

	Position [m]			Orientation [rad]		
	x	y	z	roll	pitch	yaw
Seed 1	0.0869	0.730	-0.1897	$-\frac{\pi}{2}$	0	$\frac{11\pi}{12}$
Seed 2	0.0869	0.730	-0.1897	$-\frac{\pi}{2}$	0	π
Seed 3	0.0	0.90	-0.10	$-\frac{\pi}{2}$	0	π

Table 6.7: Seeds used for the non-linear problem.

result a average position of the camera.

$$P_{base}^{cam} = P_{base}^{link} - S_{base}^{cam} \cdot P_{cam}^{link} \tag{6.1}$$

	Method	Position [m]			Orientation [rad]		
		x	y	z	roll	pitch	yaw
Seed 1	Result	0.1308	1.040	0.2141	-1.8846	0.3717	1.9934
	Average	0.0845	0.7705	-0.010			
Seed 2	Result	-0.2659	0.9928	0.045	-1.7483	-0.5772	2.147
	Average	-0.0281	0.7579	-0.0615			
Seed 3	Result	-0.0264	1.0466	0.151	-1.8424	-0.2995	2.0552
	Average	0.055	0.7647	-0.0446			

Table 6.8: Results from the optimization problem and the average camera position, for every seed.

(3) Using multiples transformations

Finally, the last approach which use all the information given by the camera, both position and orientation. So it has to use several transformations: (1) from `base_link` to link; (2) from link to modeled object and (3) from modeled object to camera. The first one is given by the simulator. Then, the second one is estimated by the user placing the object on a specific position and orientation respect to the link. Finally, the last one is known from the information given by the camera. It should be stated that for this test has been used the `l_wrist` link which is the one placed more precisely.

Therefore, the information used is shown in Table 6.9 which should be convert to transformation matrix (rotation plus translation), as eqs. (6.2) to (6.4).

Target	Reference	Position [m]			Quaternion			
		x	y	z	x	y	z	w
Modeled object	Camera	-0.1461	0.1831	0.6645	0.5484	0.5190	-0.4372	0.4886
`l_wrist`	Modeled object [6]	0	0	0	0	0.7071	0	0.7071
`l_wrist`	`base_link`	0.1604	0.2048	-0.3114	-0.1014	-0.7982	-0.15579	0.5731

Table 6.9: Information used to approach the camera position using multiples transformations.

$$T_{obj}^{cam} = \left(\begin{array}{ccc|c} 0.0790 & 0.9965 & 0.0275 & -0.1461 \\ 0.1420 & 0.0161 & -0.9897 & 0.1831 \\ -0.9867 & 0.0821 & -0.1402 & 0.6645 \\ \hline 0 & 0 & 0 & 1.0000 \end{array}\right)^{-1} \tag{6.2}$$

$$T_{link}^{obj} = \left(\begin{array}{ccc|c} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right)^{-1} \tag{6.3}$$

$$T_{base}^{link} = \left(\begin{array}{ccc|c} -0.3227 & 0.3404 & -0.8832 & 0.1604 \\ -0.0167 & 0.9309 & 0.3649 & 0.2048 \\ 0.9464 & 0.1325 & -0.2947 & -0.3114 \\ \hline 0 & 0 & 0 & 1.0000 \end{array}\right) \tag{6.4}$$

Finally, the multiplication of the three transformation matrix as in Equation (5.8), give as a result the final transformation, Equation (6.5), which contains the position of the camera and its orientation. Therefore, the final result of camera position and orientation is the one shown in Table 6.10.

$$T_{base}^{cam} = \left(\begin{array}{ccc|c} -0.8621 & 0.3535 & -0.3632 & 0.2110 \\ 0.4367 & 0.1545 & -0.8862 & 0.8292 \\ -0.2571 & -0.9226 & -0.2876 & 0.0110 \\ \hline 0 & 0 & 0 & 1.0000 \end{array}\right) \tag{6.5}$$

[6] The offset position of the link with modeled object is already applied when it is calculated the position of the object, see Section 5.1

	Position [m]			Orientation [rad]		
	x	y	z	roll	pitch	yaw
Camera	0.2110	0.8292	0.0110	-1.8846	0.3717	2.7525

Table 6.10: Camera position and orientation using multiple transformations.

6.5 Error produces by the camera positioning

Finally, the last experiment is made to know the influence of the precision positioning the camera has on the total error of the grasping. Therefore, here it is compared the position of the object calculated by the `romeo_grasper` package with the position, given by the simulator, of a link moved to where the object was. Both position are in robot reference, so they should be the same. However, move the link to the specific place has an error which is estimated around ± 3 cm. In this case has been chosen the `l_wrist` link because it is the easiest link to move. Moreover, to be as precise as possible has been used the camera position, as pre-known, result of the experiment in Section 6.3. The following Table 6.11 shows the tests done:

	Position [m] by `romeo_grasper`			Position [m] by simulator			Distance [m]
	x	y	z	x	y	z	
Case 1	0.225	0.296	-0.141	0.270	0.243	-0.154	0.0704
Case 2	0.272	0.016	-0.089	0.244	-0.023	-0.1517	0.0790
Case 3	0.350	0.097	0.030	0.340	0.040	0.037	0.0583
Case 4	-0.033	0.168	-0.246	-0.078	0.19	-0.298	0.0725

Table 6.11: Results of error produced by the camera positioning.

Chapter 7

Discussion

On this Chapter it is discussed every experiment done before to figure out what is the meaning of the results.

7.1 Simulation results

Then, here it is discussed the experiment about the solutions given by the inverse kinematic solver to go to the goal positions. In order to see the results see Section 6.1.

Firstly, about the `graspPlan` function, apart from the minor influence of the used computer, the success of this movement depends on the planning time. That can be seen observing that with a planning time of 60 seconds almost all the cases give a solution. However, these solutions have a big mean, may be that it is caused by the fact of finding the way to solve the most difficult grasps. Moreover, the real planning time it seems to depend on the computer, where the one with better graphic card is faster. So probably the unknown task is using this component, may be to simulate the grasp in the simulator. An interesting point of this is that the real planning time it seems to not be related with the planning time set by the user.

Secondly, about the `reachAction` it seems like the approximated solutions are by far more easy to calculate, but they have a large variability and, in comparison with the solution inside tolerances, they are more imprecise. Therefore, the aim should be to always find a solution inside tolerances. As said in the experiment, it is better to use small planning times but with a lot of attempts. That's sounds logic seeing that the solutions are usually calculated with an small amount of time. Therefore, it has a low probability to get a solution after a few seconds, so it is better to make another attempts

with a bit higher goal tolerance. Finally, remark that the influence of the computer is important here, because using one or another can lead to not have almost any success or the half of the tries. Moreover, this affects also the precision of the approximated solutions which gives a better performance with a better processor.

Furthermore, as described in Section 2.5, the `graspPlan` is a more complete functionality because it adapts to the object orientation. However, it last so much time and is less accurate than the `reachAction` with solutions inside tolerances. Therefore, for a typical use of not complicated orientations it should be used `reachAction` and for the rest `graspPlan`. Moreover, if the result doesn't have the desired precision the plan can be remade and give another result with the user service if it is not in automatic execution, see Section 5.3.2.

Finally, summarising the main points of this experiments are:

- The `graspPlan` needs a large planning time to succeed.
- It should try to get solutions inside tolerances, so use low planning times and high amount of attempts, but the success rate can depend on the computer.
- It is better to use `reachAction` for simple orientation (vertical) and `graspPlan` for the rest.
- If the result is not enough accurate remake the plan with the user service.

7.2 IKFast on Romeo

Below are explained the results and possible causes of this results. Unfortunately, at least by now it has not been possible to implement the IKFast solver on Romeo. The reason is because when the C++ code has been tried to generate, it failed. However, it should look for the reason of this fail problem.

Reading the OpenRAVE forum [**?**] and the home page [11], I figure out that an important thing to make easier to the program to generate the code is to find the intersecting and non-intersecting axes. That's done looking on the robot model, but if it has some deviation, too much decimals or may be the orientation of the joint, among some possible causes, it is more difficult to find these axes. Therefore, a way to be able to implement the IKFast solver on Romeo is to try to make easier to the code generator its work.

7.3 Camera positioning

The experimental way of getting the camera position in Section 6.3 is somehow accurate, at least it seems to not have a huge precision error. However, the visual positioning has in general a huge difference with the initially got.

Looking at the results it seems like the lack of precision of the `LElbow` and `LShoulder` affects to the final result. Because in the first approach using only one link the closest to the real camera position is the one from `l_wrist` which may be using the right orientation could lead to a more accurate result. However, the approaches with this method from `LElbow` and `LShoulder` are too far to be used.

Moreover, the error is even worse for the method using the optimization problem which non of the results could be used. Although, the precision error could have some of the responsibility for this result, it is important to state that another factor is the weights for the minimising function which currently are all the function with the same weight, but the results change drastically with other weights.

Finally, the last one, using multiple transformations, is the most logical one to use, for it simplicity and the use of all the information with only recognising an object. Unfortunately, the position that gives as a result is not enough accurate, but at least the orientation is close to be the right one. Summarising, the visual positioning if it is wanted to be used has to be improved, between the three methods the one which is more worth to make an effort to improve is the last one.

7.4 Error produced by the camera positioning

Chapter 8

Future works

8.1 Implementation new inverse kinematics

Currently, there is some variability in the error produced by the inverse kinematic solver KDL to go to the position of the object. Moreover, it depends a lot on the planning time which is too big to allow to properly interact with the robot. Therefore, the implementation of a new IK solver should be a great improvement for the work. Moreover, currently the picking action is not working well due to the IK solver doesn't find a good picking plan, so with this new solver it is supposed that this should be solved.

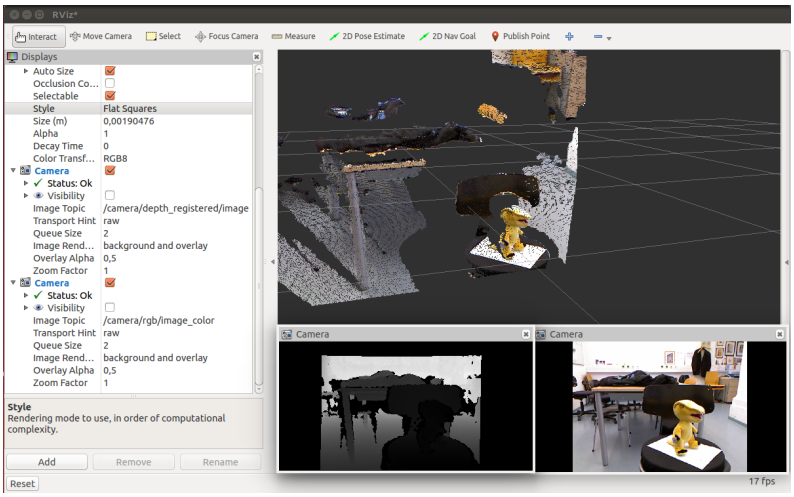
The first recommendation would be to implement the IKFast from OpenRAVE, because it is a close form solver and so it very fast and precise. As seen in Section 6.2, currently is not working, but may be can solve it editing the URDF model trying to make it easier or changing the initial pose. Also is recommended to read this issue [2], where there is a lot about it discussed.

In case that the previous solver could not be used, there is an other option used in [9] named Metapod from LAAS, which is also a numerical solver. Therefore, probably it has almost the same problems as KDL, but at least has been already tested with Romeo.

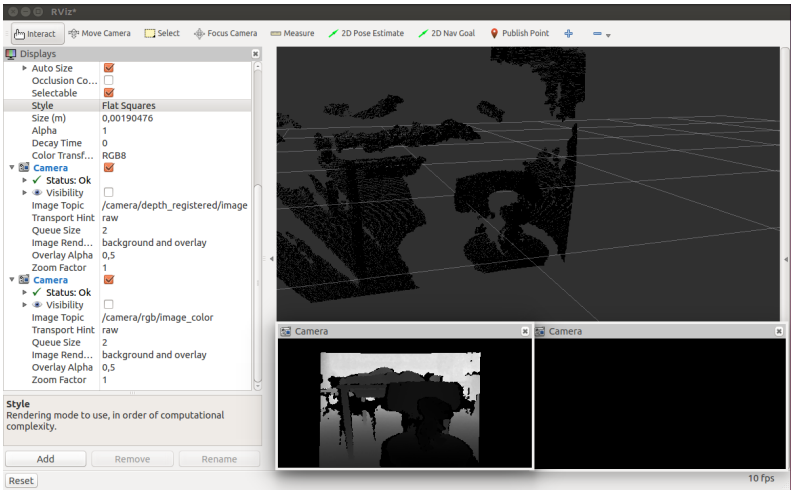
8.2 Improve use of RTM

The RTM software is a very powerful tool to recognise objects and tracking them. However, it has to be complemented with the convenient camera to get the best performance. Currently, the Kinect camera, which sends the images to ROS through OpenNi framework, gives some problems. On one hand, after some seconds or minutes the RGB

camera starts to send partial or full black images, example of it in Figure 8.1. Therefore, the RTM software cannot be used properly. On the other hand, the OpenNi nodes after a while usually fail due to memory or pointers errors and then they are automatically shut down.



(a) Normal performance



(b) While is not working

Figure 8.1: Kinect problem, RGB camera sending black images.

Moreover, RTM process the images with a kind of SIFT on the RGB image, so having a good quality of the RGB camera it could lead to a better performance. Unfortunately, the Kinect used is the worst in terms of RGB camera in comparison with the Xtion and the RealSense, as shown in Table 8.1. Therefore, to improve the performance of the recognition would be great to use RealSense or Xtion. On one hand, currently, the use of RealSense is possible,[1] but the object must be modeled with another camera because RealSense is not supported by RTM-Toolbox and that decrease dramatically the confidence of the tracking. Therefore, this option would be great if the RTM-Toolbox is

[1]The `use_realsense` argument enables to launch the `romeo_grasper` using the RealSense camera.

improved and gives support to RealSense. On the other hand, for the Xtion there are two possibilities. First, with an external one as it has been done with Kinect in this work. Second, using the one on the Romeo cap. For this last one, only could work with objects on a specific and small range because of the camera range. However, it would remove some error produced by the camera positioning because the position of this camera is established in the model. Moreover, in order to solve the problem with dark images may be a possibility is to try to execute ROS with OpenNi inside the robot.

Camera	RGB resolution	Depth resolution	Range
Microsoft Kinect v1 [28]	640x480	320x240	0.5-4.5 m
ASUS Xtion PRO Live [8]	1280x1024	640x480	0.8-3.5 m
RealSense SR300 [15]	1920x1080	640x480	0.2-1.5 m

Table 8.1: Comparison of the RGB-D cameras: Kinect, Xtion and RealSense.

8.3 Object identification

Although, it is not explained before how it works, RTM can recognise several objects at once with the Recognition feature. These objects should be in a specific database. A detailed description of this feature can be found in the tutorial of V4R ROS wrappers repository [32]. There are several factors of the object recognition that can improve the performance of the `romeo_grasper` package:

- Provides the centroid of the cluster, which could be used to give to the package where it have to make the grasp.
- Provides a bounding box of the object, which is how the object is represent in the simulation by now. So this would make to don't need to use a custom bounding box made by the user.
- Provides a point cloud of the model transformed into camera coordinates. This information could be used for the AGILE grasp to grasp the desired object in a better way.
- Identifying several objects at one could be used to implement a better way to use the visual positioning of the camera. Using a different identifier for every element could be possible to know the position of every link at once. Therefore, it makes the process more robust to find the camera position and orientation.

The only problem of this way of get the position of the object is that it is not optimal for a real-time tracking. This feature works as a service, so in order to work with it, the

service should be called periodically to know always where are the object.

8.4 Improvement of visual camera positioning

As seen in the experiments, the positioning of the camera identifying parts of the robot doesn't give good results. Above has been explained an improvement for this, being able to identify every link at one. Therefore, implementing that and also QR-Codes, as in [9], for every link, it could give a better result. These method has two main advantages: (1) a QR-Code is easy to recognise by SIFT because it has a lot of interesting points and (2) these QR-Codes are fixes respect to the robot, so they can have their own offset precisely measured. The main disadvantages are that the QR-Codes should be visible always when the package is launched and that after the initialization they are not used any more and may occlude part of the environment.

8.5 Fix the Romeo model

Currently, the grippers in the Romeo model are posed in a wrong position, see Figure 8.2. However, at the moment, this is not used because the grasp position is specified by `romeo_grasp_data.yaml` of the `moveit_simple_grasps` package. Then, the grasp position is set adding an offset from the WristYaw_link position. Therefore, should be great to make the model more standard and have the gripper in the right place and set the grasp from the gripper itself.

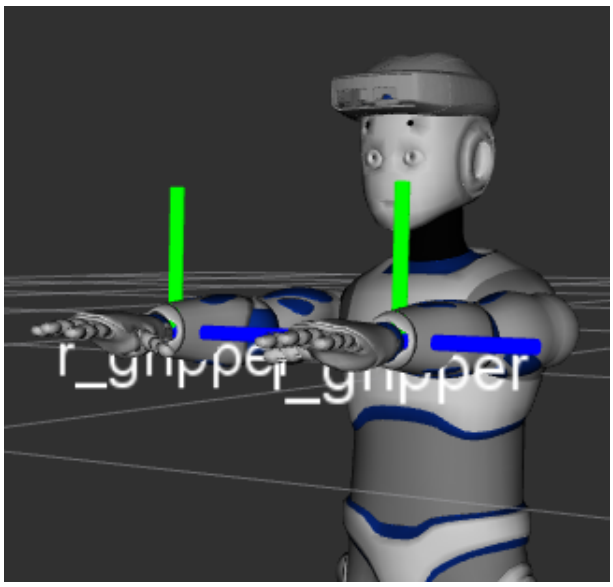


Figure 8.2: Current position of grippers in Romeo model.

8.6 Implementation of AGILE grasp

Once the first approach of grasping works perfectly, it is time to implement a more advanced grasping process. As explained in Section 2.3, the AGILE grasp use machine learning to detect a antipodal grasp pose on novel objects, using as input a point cloud and the geometric parameters of the robot hand. However, the implementation could be made in two different ways. First, it could use the general point cloud of the camera which it will have some occluded parts of the object. Second, if the recognition object, explained in Section 8.3, is implemented then use the point cloud of the model transformed into camera coordinates which it won't have occluded parts because it is a combination of the modeled object and the real object.

8.7 Dependencies of packages

A package is more flexible and can be reused in other fields as independent is of other packages. On one hand, it would be great to make this package as general as possible, in order to be able to be used for other robots. That could be made if the `romeo_moveit_actions` have the same functions of the moveit actions of others robots, then it is supposed that it should not be very difficult with the right definitions for the building part. On the other hand, making the code be independent of the recognition software could lead to used this grasping package in a wider range of fields.

Chapter 9

Conclusion

Bibliography

[1] Aldebaran 2.1.4.13 documentation. URL http://doc.aldebaran.com/2-1/.

[2] Romeo robot repository issue, 2016. URL https://github.com/ros-aldebaran/romeo_robot/issues/10.

[3] Aldebaran. C++ SDK Installation — NAO Software 1.14.5 documentation, . URL http://doc.aldebaran.com/1-14/dev/cpp/install_guide.html#requirements.

[4] Aldebaran. Getting Started — qiBuild 3.11.3 documentation, . URL http://doc.aldebaran.com/qibuild/beginner/getting_started.html.

[5] Aldebaran. ROS-Aldebaran repository, . URL https://github.com/ros-aldebaran/.

[6] Aldebaran. ROS-NAOqi repository. . URL https://github.com/ros-naoqi.

[7] A. Aldoma, F. Tombari, J. Prankl, A. Richtsfeld, L. Di Stefano, and M. Vincze. Multimodal cue integration through Hypotheses Verification for RGB-D object recognition and 6DOF pose estimation. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 2104–2111, 2013. ISSN 10504729. doi: 10.1109/ICRA.2013.6630859.

[8] ASUS. ASUS Xtion PRO LIVE Spec. URL https://www.asus.com/us/3D-Sensor/Xtion_PRO_LIVE/specifications/.

[9] Giovanni Claudio, Fabien Spindler, and François Chaumette. Grasping by Romeo with visual servoing. In *Journées Nationales de la Robotique Humanoïde, JNRH*, Nantes, France, 2015. URL https://hal.inria.fr/hal-01159882.

[10] Dave Coleman. MoveIt Simple Grasp. URL https://github.com/davetcoleman/moveit_simple_grasps.

[11] Rosen Diankov. OpenRAVE | Home. URL http://openrave.org/.

[12] Rosen Diankov. Automated Construction of Robotic Manipulation Programs. *Architecture*, Ph.D.:1–263, 2010. ISSN 978-1-124-53547-0. doi: isbn9781124535470. URL http://www.programmingvision.com/rosen_diankov_thesis.pdf.

[13] Tully Foote and Mike Purvis. REP 103 – Standard Units of Measure and Coordinate Conventions (ROS.org), 2010. URL http://www.ros.org/reps/rep-0103.html#suffix-frames.

[14] Inria Lagadic group. Romeo humanoid robot grasping demonstration v1, 2014. URL https://www.youtube.com/watch?v=kz1Ob0Ks554.

[15] Intel(R). Intel RealSense Camera. URL https://software.intel.com/en-us/realsense/devkit.

[16] Intel(R) and Séverin Lemaignan. Repository Intel® RealSense™ Technology - ROS Integration, 2016. URL https://github.com/severin-lemaignan/realsense/tree/indigo-devel/camera.

[17] Khronos Group. COLLADA - 3D Asset Exchange Schema. URL https://www.khronos.org/collada/.

[18] Sophie Li. Installing Kinect drivers on Ubuntu 14.04 and ROS Indigo – sophie's blog, 2015. URL http://blog.justsophie.com/installing-kinect-nite-drivers-on-ubuntu-14-04-and-ros-indigo/.

[19] Par Paul Loubière. Aldebaran a fini de jouer avec Nao le petit robot, apr 2015. URL http://www.challenges.fr/entreprise/20150402.CHA4548/aldebaran-a-fini-de-jouer-avec-nao-le-petit-robot.html.

[20] Natalia Lyubova. Moveit Simple Grasp, 2016. URL https://github.com/nlyubova/moveit_simple_grasps/tree/romeo-dev.

[21] MoveIt. Creating a custom IKFast Plugin - MoveIt IKFast documentation, 2013. URL http://docs.ros.org/hydro/api/moveit_ikfast/html/doc/ikfast_tutorial.html.

[22] Penton. Motors and Drives Leading the Way for Robots. URL http://machinedesign.com/motorsdrives/motors-and-drives-leading-way-robots.

[23] Johann Prankl, Thomas Mörwald, Michael Zillich, and Markus Vincze. Probabilistic cue integration for real-time object pose tracking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7963 LNCS(600623):254–263, 2013. ISSN 03029743. doi: 10.1007/978-3-642-39402-7_26.

[24] Johann Prankl, A. Aldoma, Alexander Svejda, and Markus Vincze. Multi-session Object Modelling with an RGB-D Sensor, 2015. URL https://www.youtube.com/watch?v=UMZJQqxHF98.

[25] Johann Prankl, Aitor Aldoma, Alexander Svejda, and Markus Vincze. RGB-D object modelling for object recognition and tracking. *IEEE International Conference on Intelligent Robots and Systems*, 2015-Decem:96–103, 2015. ISSN 21530866. doi: 10.1109/IROS.2015.7353360.

[26] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software*, 2009.

[27] Robot Operating System. ROS.org. URL http://www.ros.org/core-components/.

[28] Roland Smeenk. Kinect V1 and Kinect V2 fields of view compared, 2014. URL http://smeenk.com/kinect-field-of-view-comparison/.

[29] R. Smits, H. Bruyninckx, and E. Aertbeli. KDL: Kinematics and dynamics library | The Orocos Project. URL http://www.orocos.org/kdl.

[30] STRANDS. Spatio-Temporal Representation and Activities for Cognitive Control in Long-Term Scenarios. URL http://strands.acin.tuwien.ac.at/.

[31] STRANDS. V4R Wiki, 2015. URL https://github.com/strands-project-releases/strands-releases/wiki.

[32] STRANDS. V4R ROS Wrapper repository, 2015. URL https://github.com/strands-project/v4r_ros_wrappers.

[33] Andreas ten Pas. AGILE grasp - ROS Wiki, 2015. URL http://wiki.ros.org/agile_grasp.

[34] Andreas ten Pas and Robert Platt. Localizing antipodal grasps in point clouds. *CoRR*, abs/1501.0, 2015. URL http://arxiv.org/abs/1501.03100.

Appendices

Appendix A

Software requirements and installation

A.1 Software requirements

In this study there are some software that must be installed to be able to do it. Here it is explained briefly which are these software and their installation. Firstly, a table with the software required with his version A.1.

Software	Version
Ubuntu	14.04
ROS	Indigo
Choregraphe	2.1.4.5 (1.14.5)[1]
C++ NAOqi SDK	2.1.4.5 (1.14.5)[1]
QiBuild	from installation[2]
QtCreator	recent[3]

Table A.1: Software required with his version

A.2 Romeo grasper installation

Make sure the following ROS dependencies are installed:

- `naoqi_bridge_msgs`
- `romeo_bringup`

[1]NAO robot use version 1.14.5 and Romeo use 2.1.4.5. So if you want to test first in Nao like in this master thesis you need both versions.

[2]Recommended installing using `pip` as is said in qiBuild documentation [4]

[3]As is said in NAO documentation [3]

- `romeo_moveit_config`
- `romeo_moveit_actions` but should be installed from source from the following repository: `https://github.com/ros-aldebaran/romeo_moveit_config`
- `moveit_ros_planning_interface`
- `moveit_visual_tools`
- `moveit_simple_grasps` but should be installed from source from the following repository: `https://github.com/davetcoleman/moveit_simple_grasps`
- `object_tracker_msg_definitions`
- `object_tracker_srv_definitions`

After install them and their dependencies clone the repository into a catkin workspace:

```
cd my_catkin_ws/src
git clone https://github.com/lluissalord/romeo_grasper.git
cd ..
catkin_make
```

A.3 Camera drivers installation

In order to install the Kinect camera with openNi has been used the following url [18]. For the RealSense camera has been used this repository [16] following the installation instructions. Unfortunately, the Asus Xtion camera has not been possible to use so here there is no description about the installation.

A.4 V4R installation

In order to use the RTM software it should be installed the full V4R library which can be installed from ubuntu package or from source. After that, it should be installed the V4R ROS wrappers [32].

From ubuntu package

Following these steps:

- Enable the STRANDS repositories:
  - Add the STRANDS public key to verify packages:

```
curl -s http://lcas.lincoln.ac.uk/repos/public.key | sudo apt-key
    ↪ add -
```

  - Add the STRANDS repository:

```
sudo apt-add-repository http://lcas.lincoln.ac.uk/repos/release
```

- Update index:

```
sudo apt-get update
```

- Install:

```
sudo apt-get install ros-indigo-v4r
```

From source

```
cd ~/somewhere
git clone 'https://github.com/strands-project/v4r'
cd v4r
mkdir build
cd build
cmake ..
make
sudo make install (optional)
```

V4R ROS wrappers

Finally, the wrappers should be installed from source because there are some features that are not yet implemented in the ubuntu package.

```
cd my_catkin_ws/src
git clone https://github.com/strands-project/v4r_ros_wrappers.git
cd ..
catkin_make
```

A.5 OpenRAVE installation

OpenRAVE is the software which allows to create the inverse kinematics solver IKFast. This software have some dependencies which should be installed from source and it is recommended in the same order:

- `libccd` which is a library for collision detection between two convex shapes

  `https://github.com/danfis/libccd`

- `octomap`, an efficient probabilistic 3D mapping framework based on octrees

  `https://github.com/OctoMap/octomap`

- `FCL` (Flexible Collision Library)

  `https://github.com/flexible-collision-library/fcl`

- `ODE` (Open Dynamics Engine)

  `https://sourceforge.net/projects/opende/`

After installing these dependencies, it should be installed `bullet-2.80`, the recommendation is to use the following `ppa` because is the version which works well with OpenRAVE. Another recommendation is to remember to enable shared libraries in the installations, because in some of them it is not enabled by default.

`http://ppa.launchpad.net/joern-schoenyan/libbullet280/ubuntu/`

Finally, the installation of OpenRAVE is also recommended to do it from source. Moreover, my installation I had to disable some plugins as `qtosgrave` and `fclrave`. This can be done using the command `ccmake`.

`https://github.com/rdiankov/openrave`