ASSIGNMENT 1 & 2

Students:

Gabriel Taormina

Stefano Trenti

Matteo Villani

Professor:

Emanuele Menegatti

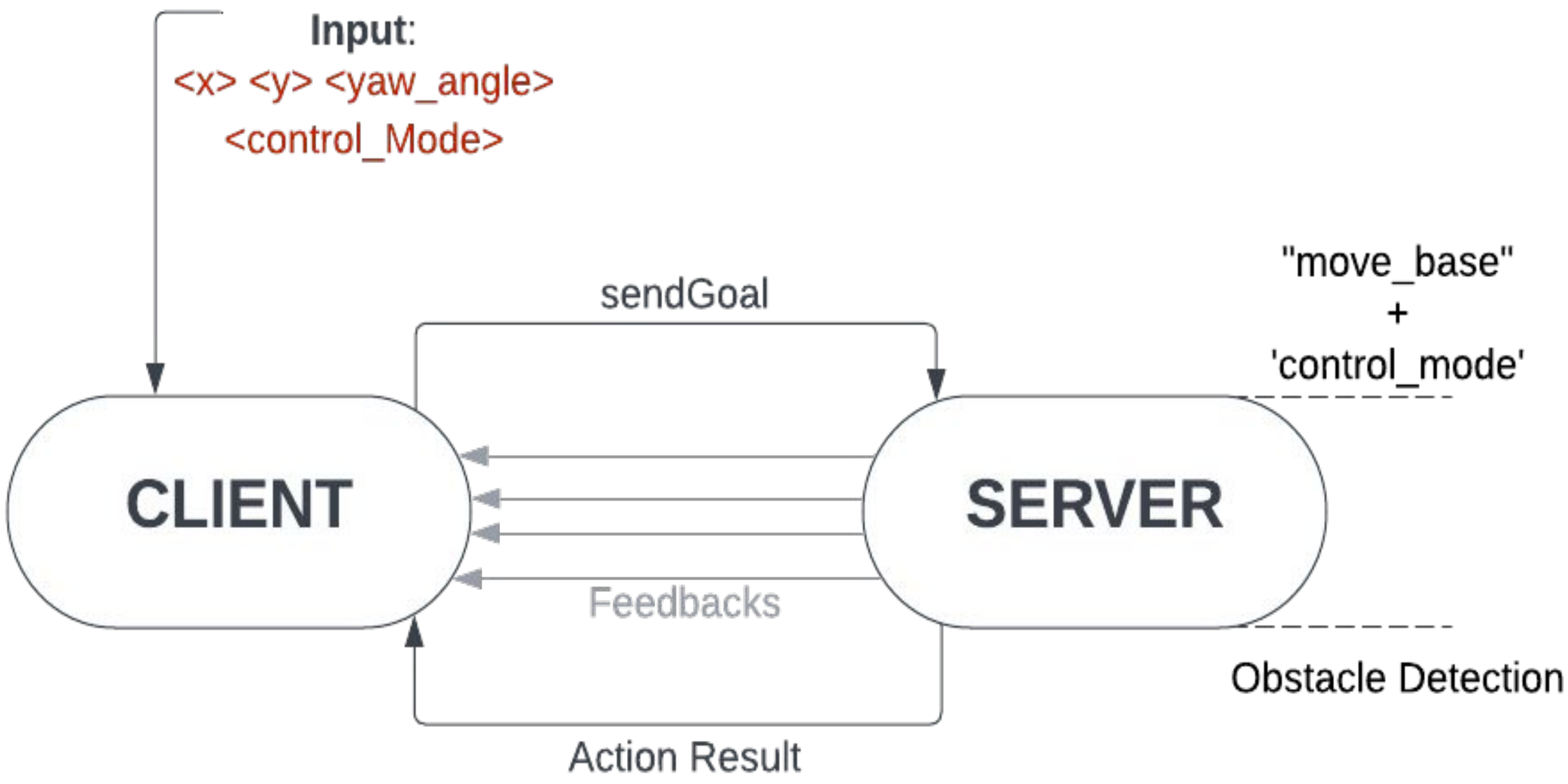
1222·2022 800 ANNI
UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

ASSIGNMENT 1

Navigation and Detection

07 Feb 2024

PROGRAM STRUCTURE



GOAL: Move into the other room and **detect the obstacles** positions

1. Command input
2. **Action Client** creates and sends goal to **Action Server**
3. **Server** connects to **"move_base"** Server
4. Server Moves the robot to the other room
5. (Update client sending feedbacks)
6. Obstacle Detection

MOTION CONTROL LAW

Navigation Stack

The robot Tiago has to reach a Pose B point starting from a Starting Pose.
It can reach the Pose B in three different modes:

- Using the control law that is already present by taking advantage of the **Navigation stack**
- Using a **motion control law** developed by us to cross the corridor **+ Navigation stack**
- Using a **motion control law** developed by us to reach the Pose B directly

```
Choose the robot's behaviour mode:
'1': only Motion-control
'2': Half Motion-Control, half Navigation
'3': only Navigation
```

The user can select the desired mode once the client is started

REACTIVE RADIAL "POTENTIAL FIELDS ALGORITHM"

Goal

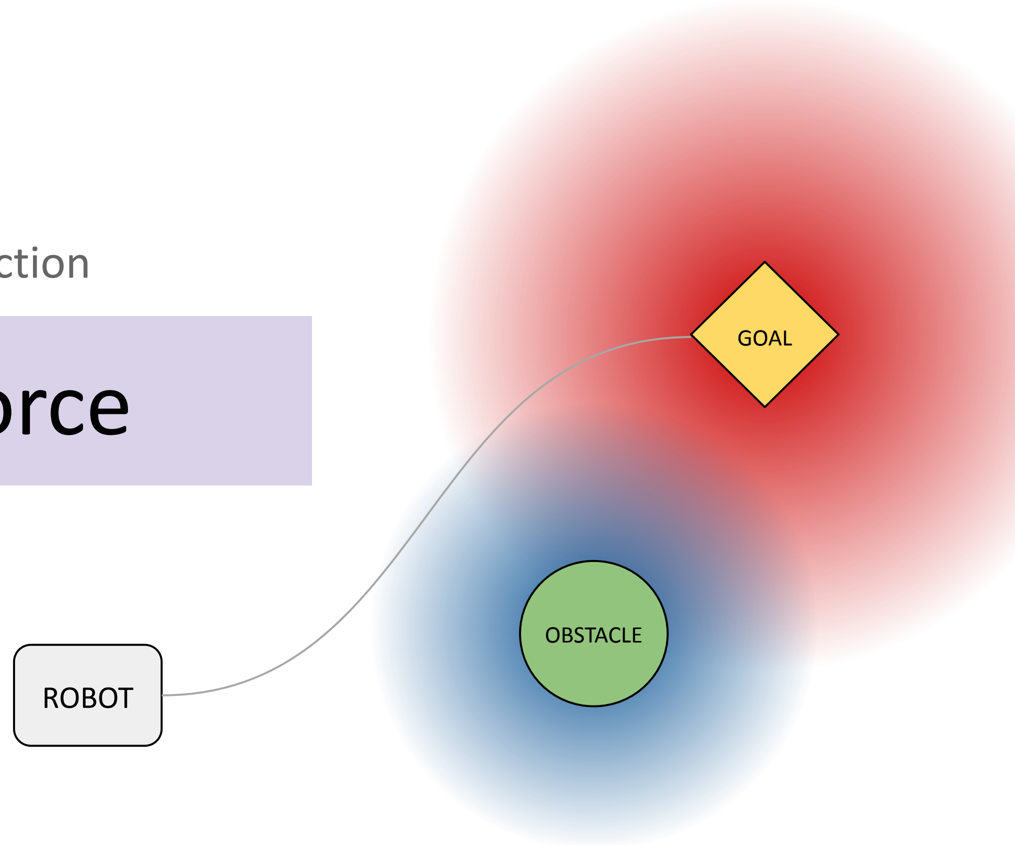
Attractive Force

Repulsive Forces

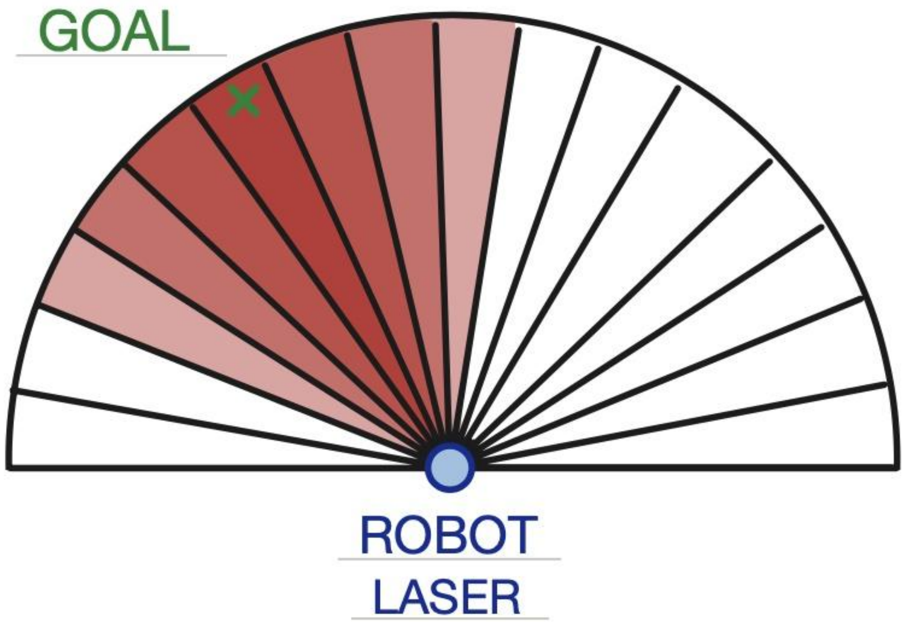
Obstacles

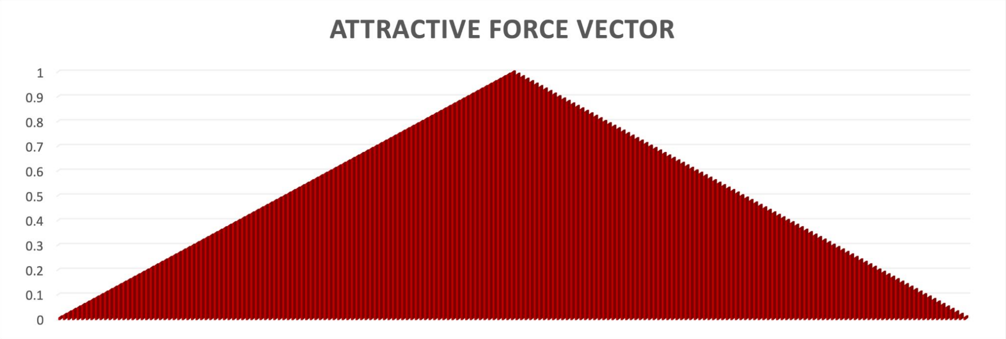
Chosen direction

Total Force



MOTION CONTROL LAW
ATTRACTIVE FORCE



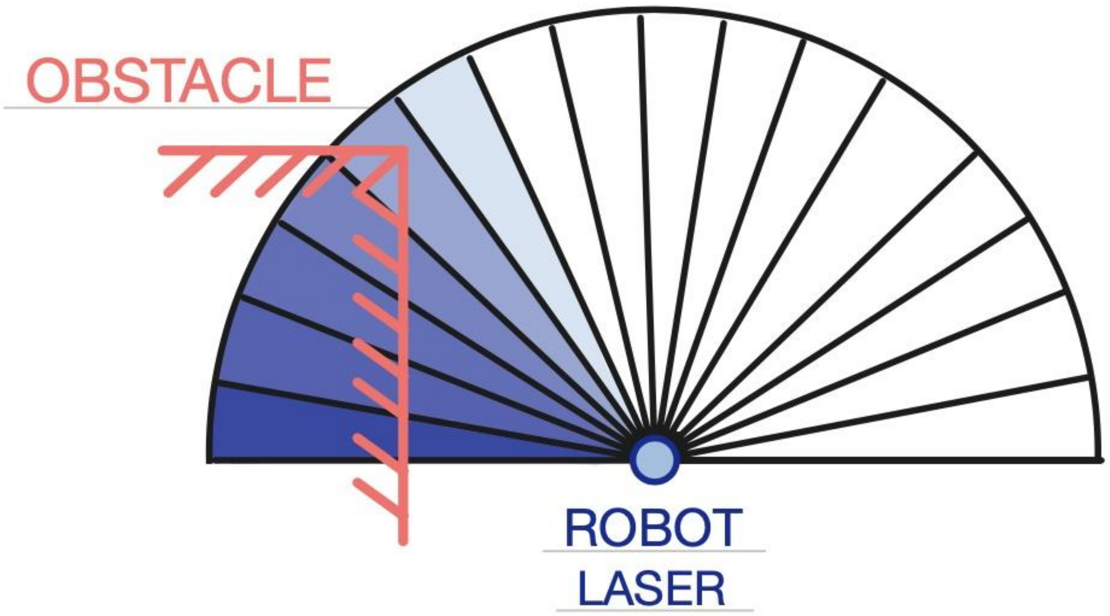


Attractive force vector:

- size = 2π × angle_increment to obtain a vector with the same angle increment but that cover 360°.
- Robot x-axis direction is in the halfway point of the array
- On the direction corresponding to the goal position the value of 1 is assigned.
- The consecutive elements of the array, on the left and right of the one corresponding to the goal, have value that decreases by a constant k = 0.01 for every new element.

MOTION CONTROL LAW

REPULSIVE FORCES



Repulsive force vector:

- Values depending on the laser scan readings
- If distance reading is above a threshold (free space) the repulsive force is set to 0
- If it is below the threshold then it is set to :

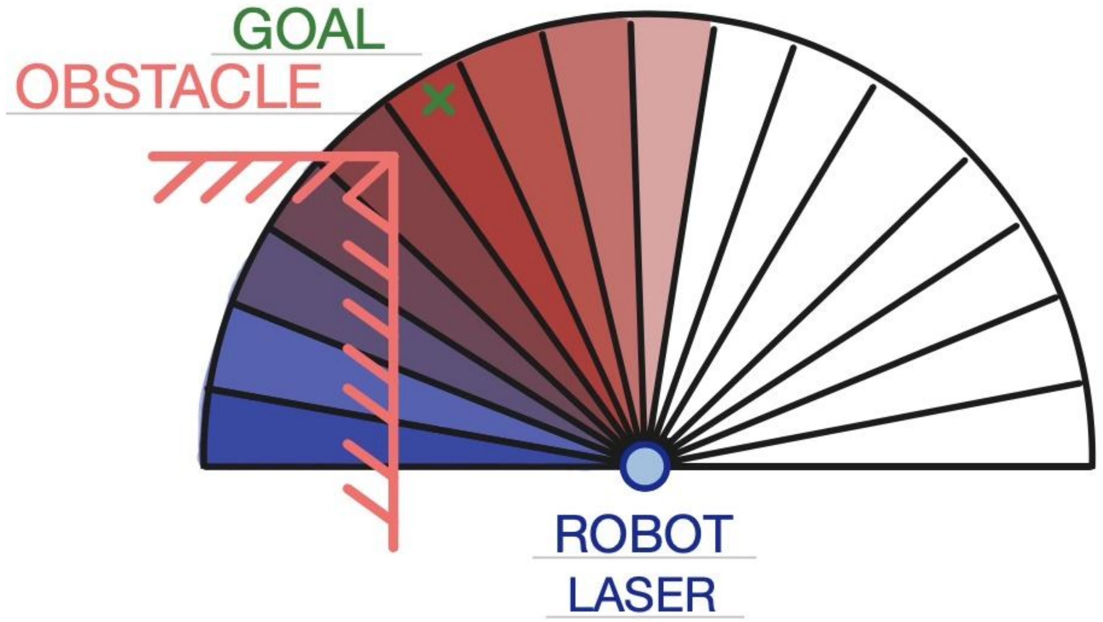
  `1 - g × distances[i]`

  where `g` is the potential repulsive gain that can be tuned.

- Some preprocessing on the distances vector is required, we consider the 8 closest neighbours to "smooth" the scan so to have a gradual fall-off from the object corners.

MOTION CONTROL LAW

TOTAL FORCE



Total force vector:

- Computed by summing the attractive force vector and the repulsive force vector
- We can finally choose the direction where to move the robot by taking the **direction with highest total force**.

Three more obstacle detection functions were created:

- two to make the robot turn avoid obstacles on it's sides pointing the robot into a safe zone
- one that considers a cone in the direction of movement the robot so that if anything is detected inside of this "danger zone" the robot avoids it

MOTION CONTROL LAW

TOTAL FORCE

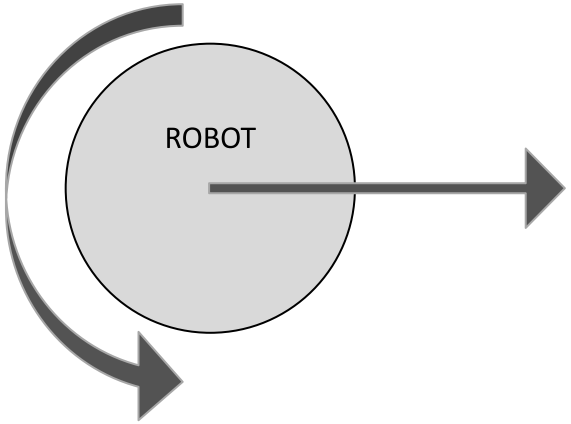
Finally the linear and angular velocities of the robot are calculated and published as in the `"/mobile_base_controller/cmd_vel"` topic as a `geometry_msgs::Twist` message.
Tiago has a **differential drive configuration**.

Linear Velocity: `v * (π – maxPotAngle)/(π * frontRange/2)`

- where `v` is the linear velocity gain, `maxPotAngle` is the chosen angle with max potential and `frontRange` is the laser detection distance in front of the robot.

Angular Velocity: `(maxPotAngle * angleVelGain)/π`

- where `angleVelGain` is a tunable gain value for the angular velocity.



OBSTACLE DETECTION

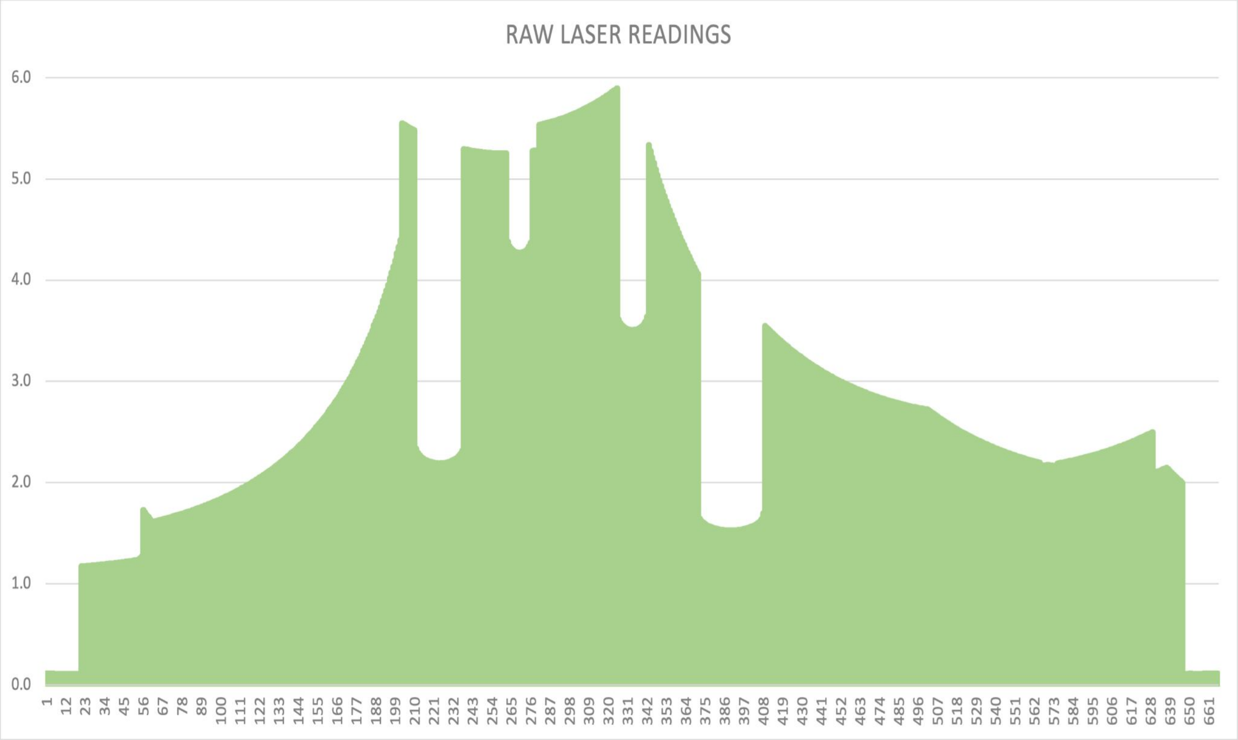
The object detection algorithm was implemented with the function:

`computeCenterPoints()`

This function performs the following steps:

- Obtains the laser scan data from the topic `"/scan"`.
- Save the distances detected from the laser scan into a vector:

```
vector<float> distances = msg->ranges;
```



OBSTACLE DETECTION

- Computes first derivative:

```
derivative[i] = distances[i-1]-distances[i]
```

leaves only discontinuities in the distance vector

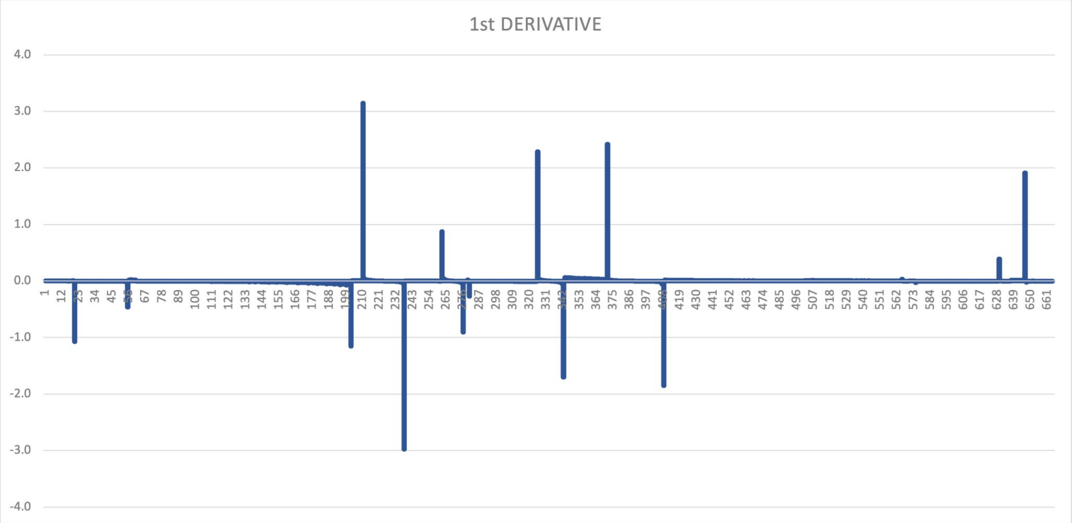
- Removing distances too close or too on the sides of the robot:

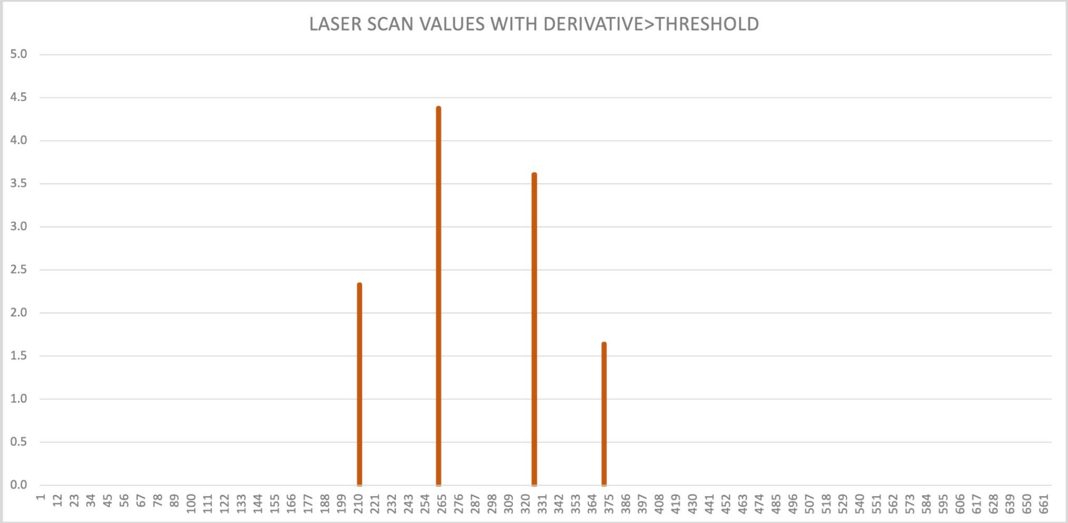
```
distances[i]<0.25 || i < 20 ||i > 646
```

- Removing distances under threshold:

```
distances[i]<0.25
```

We obtain a vector where only the right-edge of the objects have non zero value

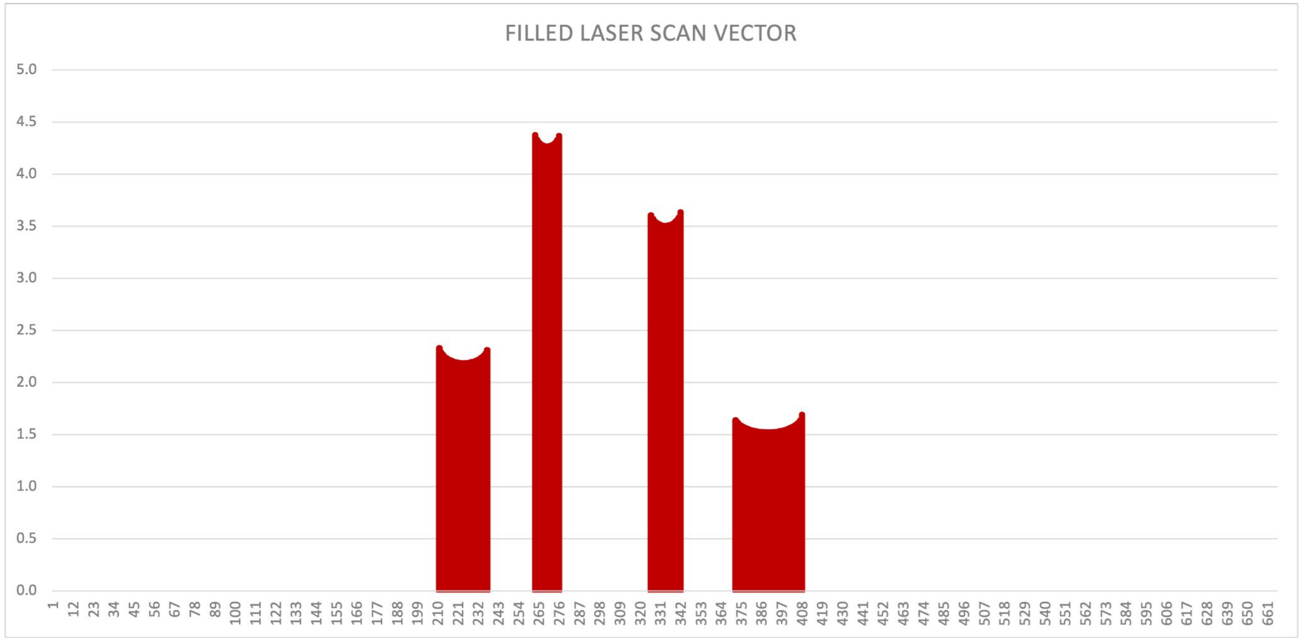


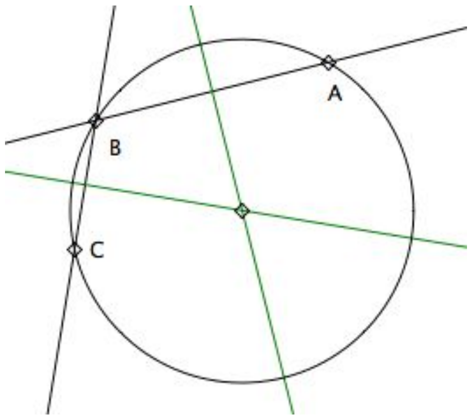


OBSTACLE DETECTION

- Keep the original distance value, starting from the right edge, until a certain threshold is met.

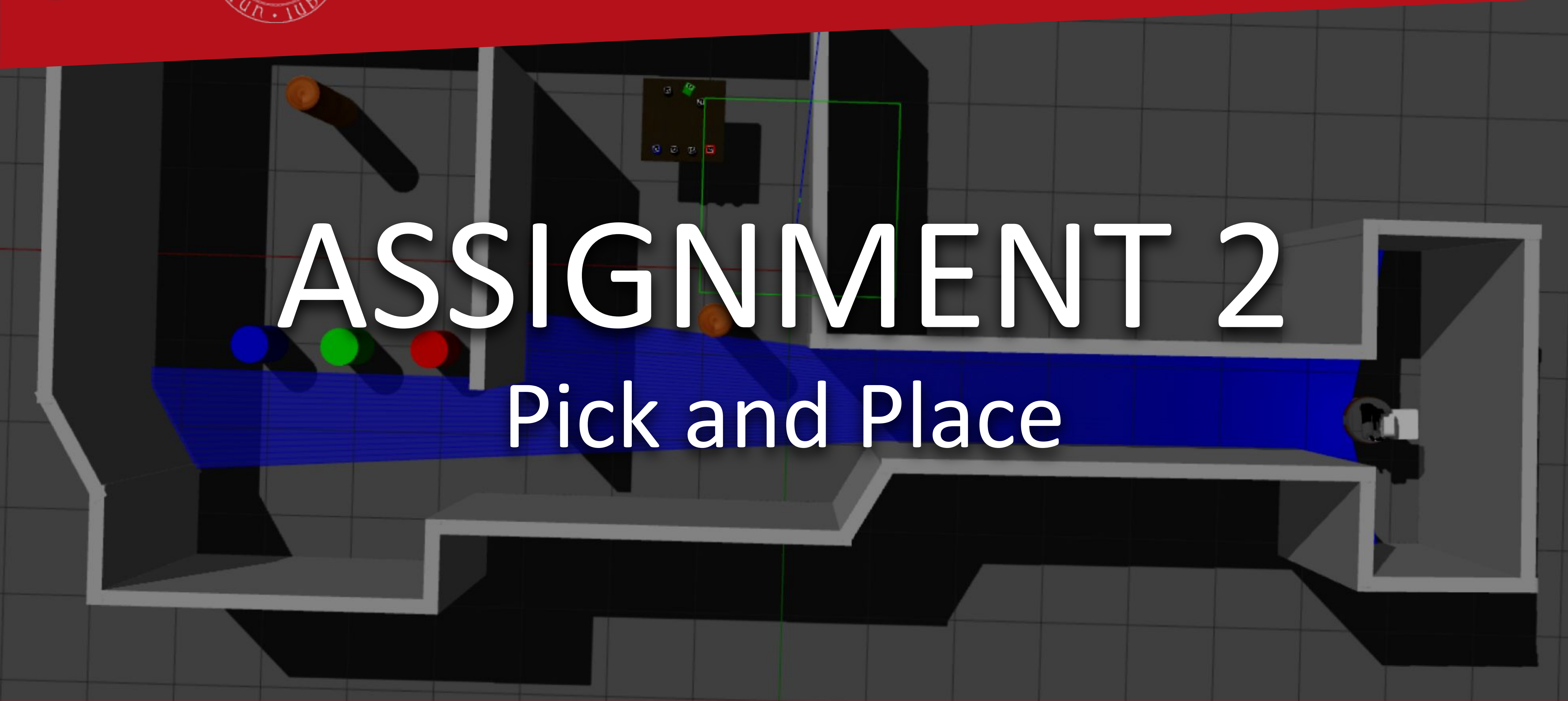
```
while(abs(distances[i+1]-beginningValue)<FILL_TRESH)
```





- Finally we take the firs, halfway, final points for every cluster of distances (corresponding to each object) and use them to obtain the center coordinates of the circle that interpolates them.

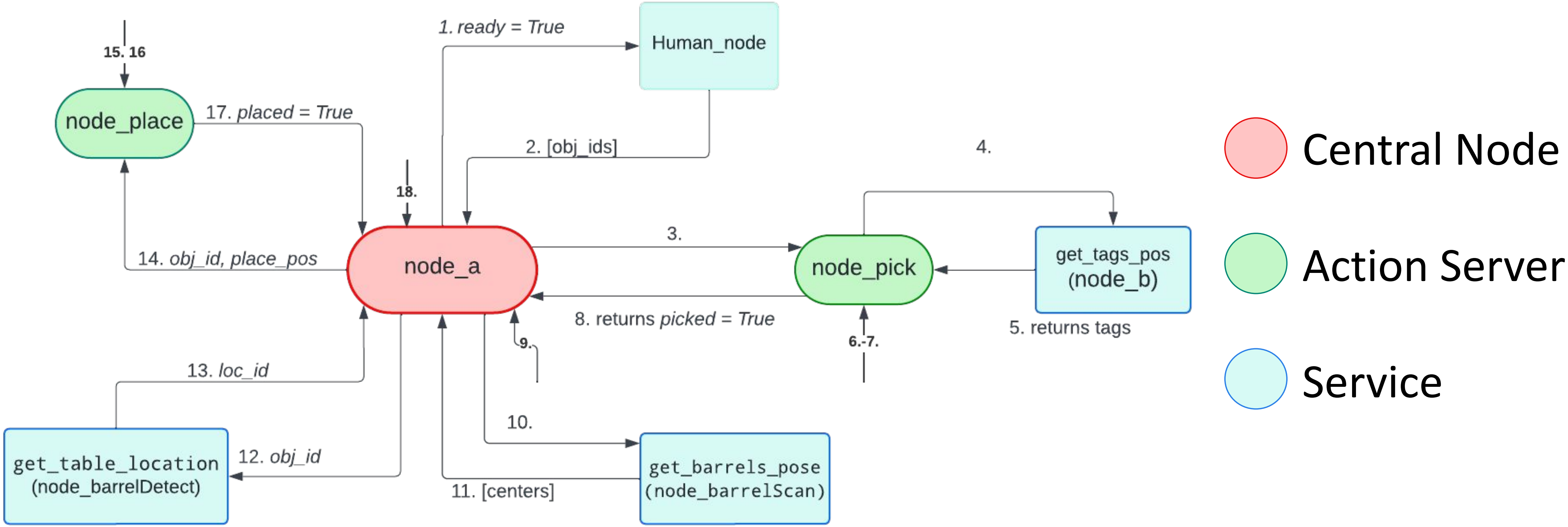
1222·2022 800 ANNI
Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione



ASSIGNMENT 2

Pick and Place

PROGRAM STRUCTURE



ROUTINE

For each of the 3 IDs the following **Routine** is accomplished by Node A:

1. Human node → **ID sequence**
2. Node Pick + Node B → **Object picked**
3. Node barrelScan + Node barrelDetect → **Target table** [EXTRA POINTS PART]
4. Node Place → **Object placed**

Implemented in `node_a` and `server`

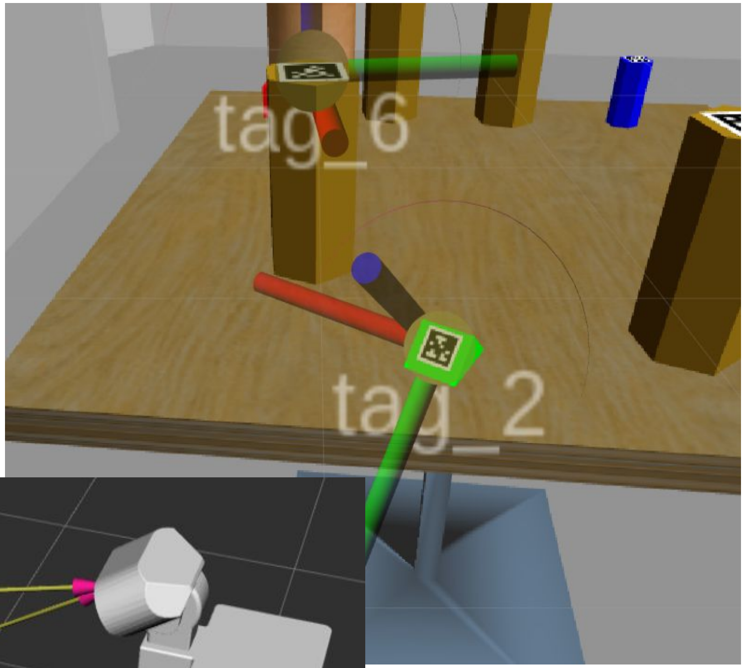
OBJECT POSE DETECTION

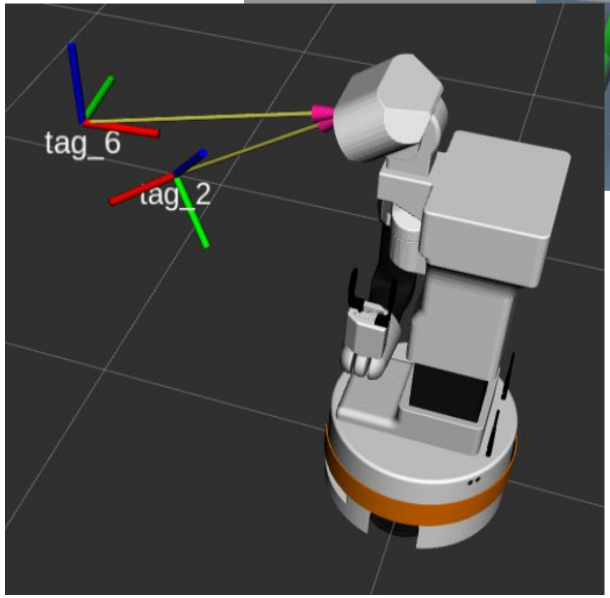
AprilTag detection

The aprilTag pose detection was implemented as a service `"get_tags_pose"`:

- **Tilts** Tiago's head toward table with custom `moveHead()` function
- **Waits** for the `apriltag_ros::AprilTagDetectionArray` message from the `"/tag_detections"` topic
- **Converts** the poses of detected tags from `"xtion_rgb_optical_frame"` to `"map"` reference frames
- **Tilts** back head

Implemented in `node_b`



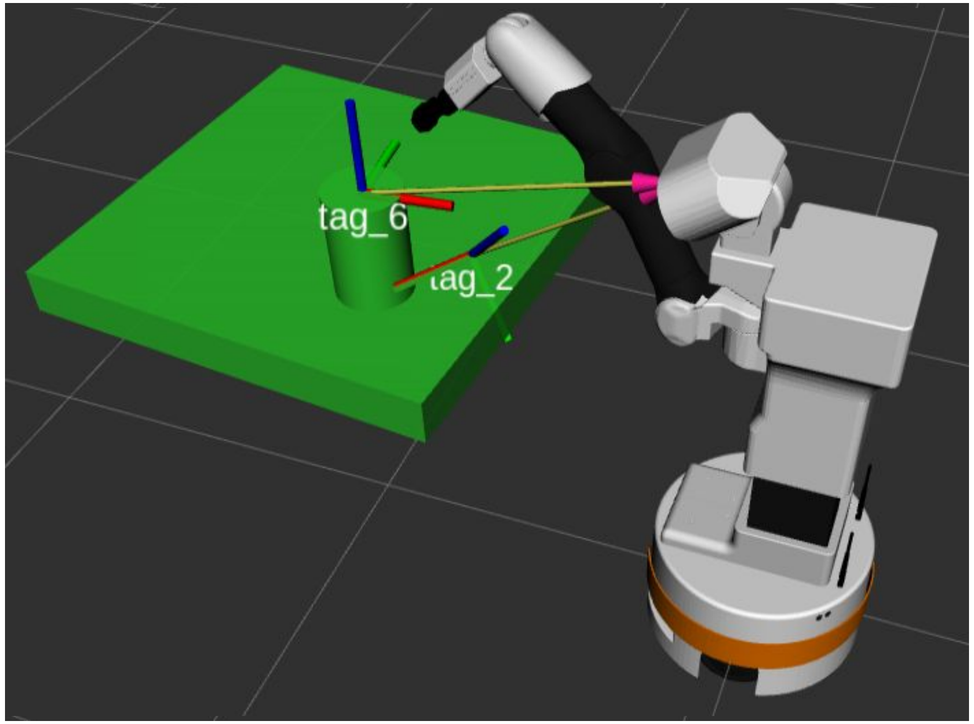


COLLISION OBJECTS

Pick Phase

Collision Objects (c.o.) are created as to avoid our robot arm crashing into objects, the steps are:

- `node_pick` calls the srv and obtains tag's **poses**
- Creates a hard-coded c.o. for the table
- Creates **cylindrical c.o.** for all tags with ID not corresponding to the **goal pickObj**
- Creates **custom c.o.** (cylindrical or cubical) depending on which object we are picking
- Add c.o.'s to `moveit::planning_interface::PlanningSceneInterface`
- After the approach phase is finished, **removes** c.o. of the object to be picked from the interface



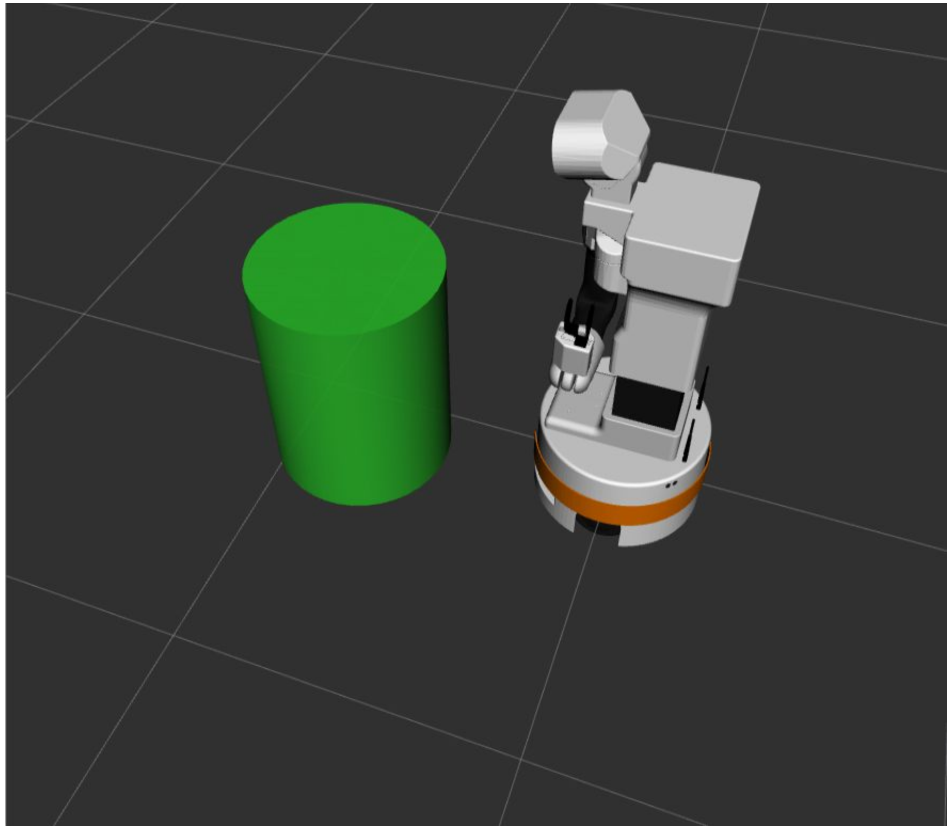
Implemented in `node_pick`

COLLISION OBJECTS

Place Phase

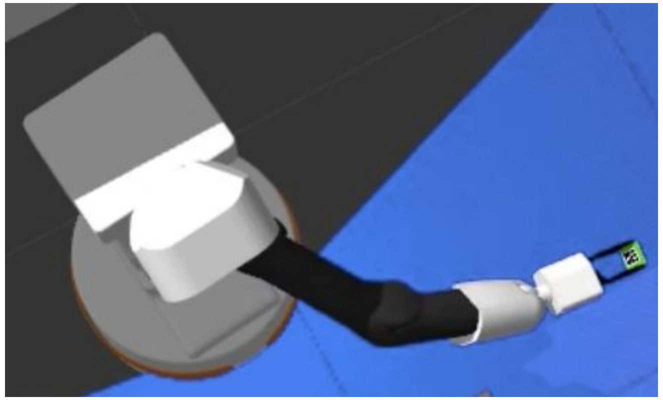
For the place phase we need the c.o. for the table where the object have to be placed:

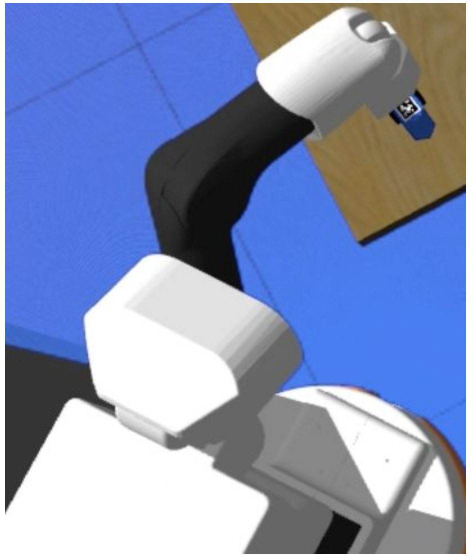
- **position** of the table is given as the **goal** of the place action function from `node_a`.
- Creates cylindrical c.o. with size slightly larger than the barrel/table and add it on the `moveit::planning_interface::PlanningSceneInterface`
- After the place phase is finished, removes c.o. of the table from the planning_scene_interface

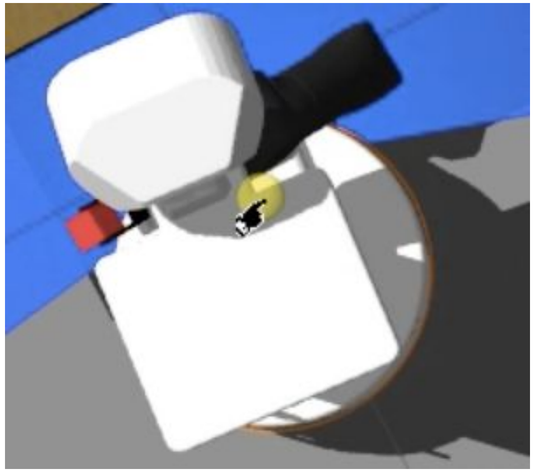


Implemented in `node_place`

PICK PHASE


Object ID 1


Object ID 2


Object ID 3

After obtaining the **poses of the various object to be picked**, using the positions and orientations with respect to the reference frame `"map"` the pick phase begins…

Implemented in `node_pick`

PICK PHASE

The pick phase consists of the following **subphases**:

- **Positioning** the arm so that it is easier to grip the object
- **Grasp** of the object
- **Attach** the object to the gripper and close it
- **Positioning** of the arm equal to the first point
- Placement of the arm in a **safe configuration for movement** within the environment

Two different functions were developed for picking: one for the red and green objects that takes advantage of the **pick()** function found in the MoveGroup Class and an **ad hoc** one for the blue object by making a plan for the **move_group** associated with the arm .

(pick from MoveGroup Class)

pick (const std::string &object, const moveit_msgs::Grasp &grasp)
Pick up an object given a grasp pose.

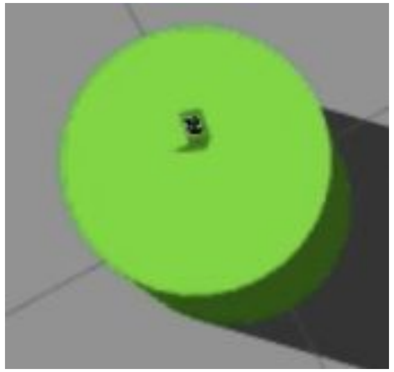
PLACE PHASE

The place phase consists of the following subphases:

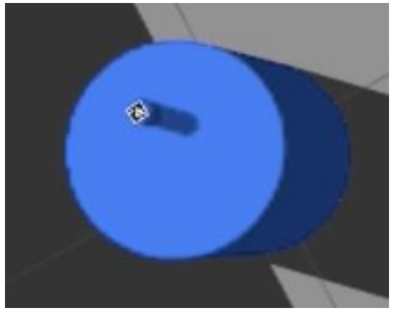
- Positioning the arm so that it is easier to place the object
- Place the object
- Open the gripper and detach the object
- Positioning of the arm equal to the first point
- Placement of the arm in a safe configuration for movement within the environment

In this case the reference frame used for the position and orientation of the object to be placed on the table is `"base_footprint"`.

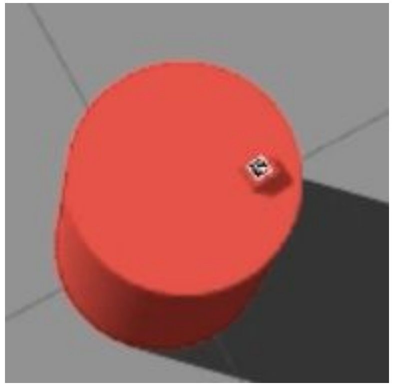
Implemented in `node_place`



Object ID 1

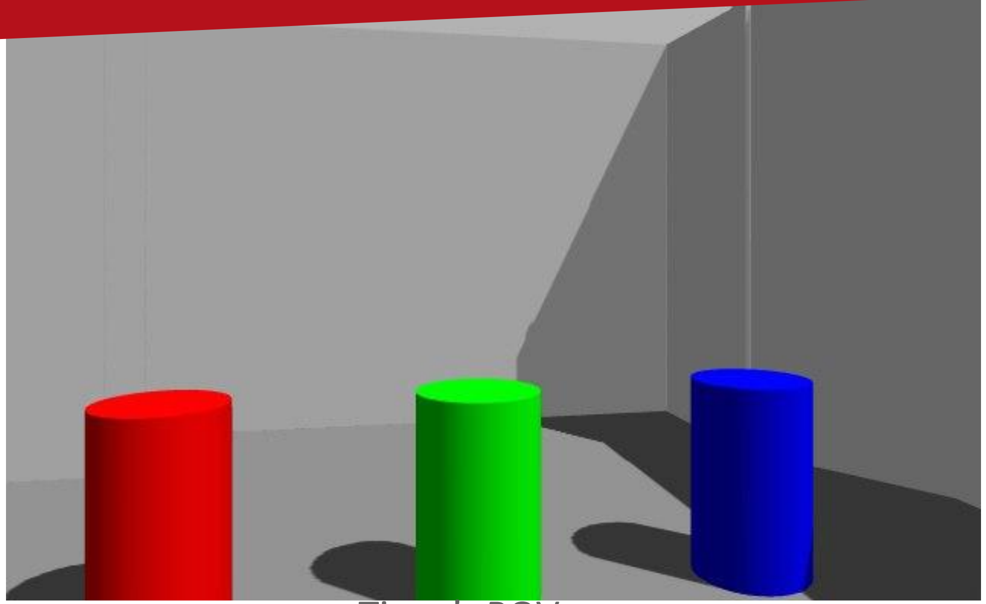


Object ID 2



Object ID 3

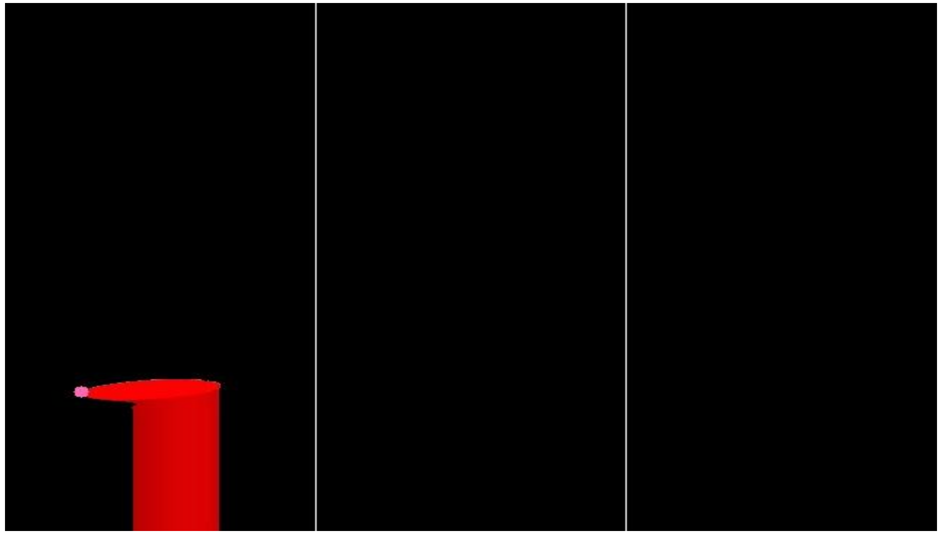
EXTRA POINTS
Automatic Docking routine


Tiago's POV

PROBLEM: Recognize which is the correct table, thus **define** the correct target position.

SOLUTION:

1. Robot moves into table room
2. Looks towards the coloured tables (Tiago's POV)

Segmented image


3. **Threshold** the image to **segment** the right table
4. The first non-black point defines the **target position** (in terms of left/center/right)

EXTRA POINTS

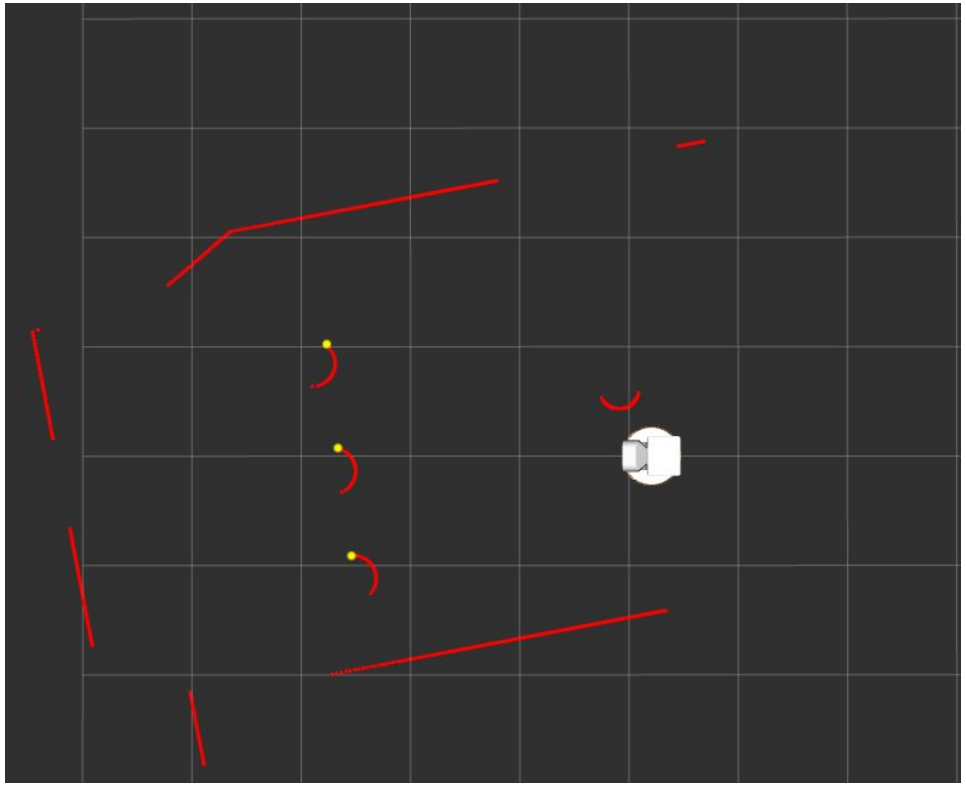
Automatic Docking routine

5. Using the same **object detection algorithm** as in assignment 1, we obtain the **center points** of the three tables

6. Use the result of the image segmentation to determine which of the three is the **correct table**

7. **Move the robot** to the obtained center table position minus a delta (in front of the table)

finally start the **place phase…**



Implemented in `node_a`

Thanks for your attention!

☺