University of Padua
Department of Computer Engineering
Intelligent Robotics

Assignment 2

by

GROUP 04
Graniello Carmine, carmine.graniello@studenti.unipd.it
Monaco Francesco Pio, francescopio.monaco@studenti.unipd.it
Qiu Yi Jian, yijian.qiu@studenti.unipd.it
Repository on BitBucket
Google Drive

November 27, 2024

Node Structure

The Robot Management `tiago_management` node was implemented as an Action Client, both the Camera `tiago_camera` and the Arm `tiago_arm` nodes were implemented using Action servers, as well as `tiago_pose` developed in *Assignment 1*, following a blueprint from the ROS tutorials [1]. This choice was motivated by the possibility of using feedback in case of problems during the execution of the task. All nodes are implemented as classes with multiple callbacks to topics of interest:

- `tiago_camera`: takes care of localizing the objects and detecting colors to find the right table for the place task. Subscribes to /tag_detections for the Apriltags, /xtion/rgb/image_raw for images of the camera, uses a tf2::TransformListener for transfomations from camera frame to robot frame;
- `tiago_arm`: takes care of the whole place/pick routines, using MoveIt! and gazebo_ros_link_attacher to perform them;
- `tiago_management`: acts as a client that coordinates the robot. Interacting with the servers to obtain the objects' order, move the robot, pick/place, fuses the information from the camera and the laser to obtain the poses for the place tables (as explained in **Place Table Pose Selection**);
- `tiago_pose`: moves the robot using both the Motion Law and `move_base`, retrieves the laser data to detect the positions of pillars.

Moving in the environment

To ensure movement with no collisions the function `getPositionMap()` was implemented to provide a sequence of waypoints for navigating around the table and reaching the room for the placement task. (Figure 1) shows the path the robot has to follow.

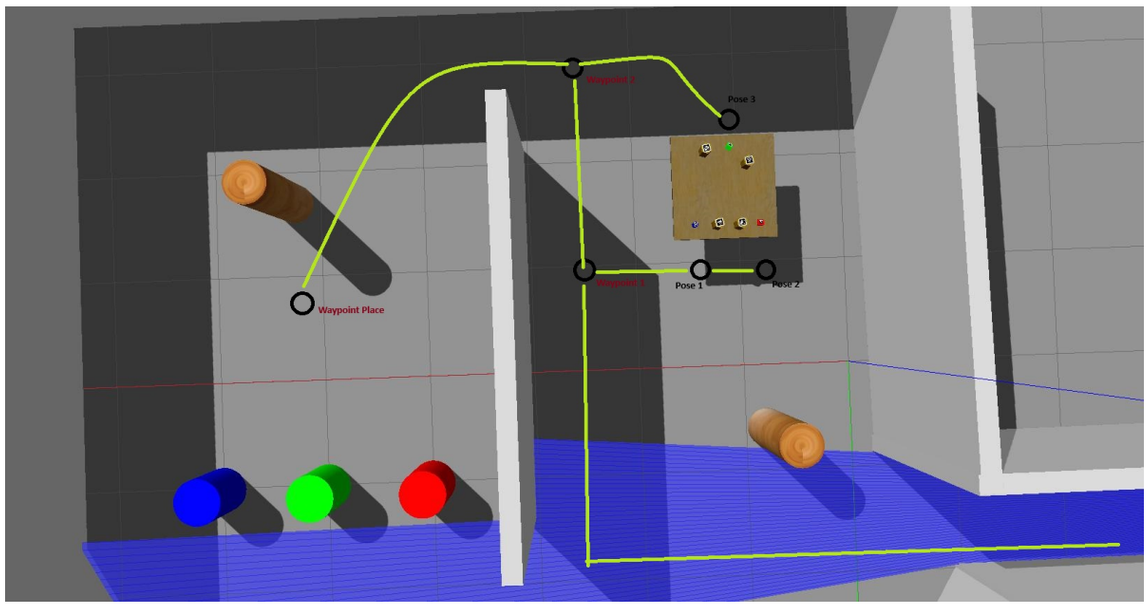

Figure 1: Map with the poses for picking (black) and for the waypoints (red). In green the path the robot is obliged to follow.

Picking the right object

To generalise our code, the robot moves around the table and uses the Camera to identify the object requested by the *human_node* at that moment. To obtain a precise localization of the objects the Camera server moves the head in 3 predetermined positions (looking straight down at the table, then slightly tilting to left and right, Figure 2) using the action interface `/head_controller/follow_joint_trajectory` and computes the mean position over 1.5 *secs* of readings of the `/tag_detections` *AprilTags* topic, the poses are tranformed in the robot reference frame (*base_footprint*) using the `.transform()` function from *tf2*, this process also ensures that the majority of the objects on the table are placed as collision objects, providing a secure working cell for the arm.

Figure 2: Poses for the head.

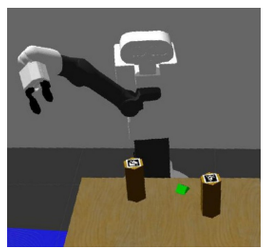

Figure 3: Safe pose.

When the correct object is recognized the arm is moved to a safe pose (Figure 3) where all the links are raised and oriented to the right with respect to the table, and the torso is slightly raised with respect to the standard position of Tiago. To perform picking, for reaching the object pose, we perform a routine of arm movements as the one suggested in the assignment, using the *MoveIt!* `.plan()` function for majority of the movements and the `.computeCartesianPath()` function during the linear movements, the move group *arm_torso* was used to have a greater number of degrees of freedom.
The grasping routine is then started using the service `gazebo_ros_link_attacher` and closing the gripper publishing a message on the topic `/parallel_gripper_controller/command`. All objects are picked from above since this allows to keep in the gripper most of the object and remove at the source collision problems during subsequent movements.
After the object has been picked following the suggested routine the arm returns to the safe pose before folding in the chassis of Tiago to enable a safe navigation in the environment.

Placing the object

To automatically find the pose for the arm to place the object, the laser routine developed for task 1 is reused. As soon as the table to be placed is reached (using the approach explained in the subsequent section), the poses of the obstacles w.r.t. the robot are obtained through the `obstacle_finder` library, the one with $min_{\ o\ \in\ obstacles}\ |o.pose.position.y|$ is the pose of the table the robot is in front of, the final position for the arm is obtained by setting an appropriate z. The routine to place the object is similar to the picking one (approaching the pose from above, linear movement to reach it, detach the object and open the gripper, linear movement to return above the object). To ensure that the arm won't collide with the placed object, during the folding in the chassis procedure, the collision object of the pillar is then enlarged in height in order to incorporate the object.

Extra-Point: Place Table Pose Selection

After picking up the object, the robot moves to an intermediate waypoint (Waypoint Place in Figure 1) in the second room. Here, the colour of the tables is recognised using the *Camera server* and *OpenCV*. After moving the head to an appropriate position we take an image from the topic `/xtion/rgb/image_raw` and save it as a `cv::Mat`.
The image is split into three sub-images that are then converted from **RGB** to **HSV**, this transformation allows a easy color detection in the hue channel, the histograms of the **H** channel are computed via `cv::calcHist` and compared to obtain a vector of colours recognised from the left subimage to right, this is possible due to the assumption that the place tables will always have some distance between themselves, producing 3 subimages with very different peaks in the histograms.

$$blue = min_{p \in peaks}\ p,\ red = \ max_{p \in peaks} p,\ blue \leqslant green \leqslant red \tag{1}$$

Obstacle detection

In order to correctly perform the object placing, the robot must know in advance where the colored pillars are in the map. To do so Tiago is placed in a waypoint (Waypoint Place in Figure 1) in the obstacles' room, in front of them. Then, an obstacle detection is performed using the same library produced for *Assignment 1*, exploiting the concept of gaps, and local minima to understand the nearest point to the robot, of the pillar itself.
Fusing this data with the color ordering done by `tiago_camera` the `tiago_management` node is able to obtain the position of the correct pillar to approach.

Problems

Building

For some days VLAB wasn't able to build the project giving as error `cannot find -lcv_bridge`, the same code was build after three days without changing the CMake, the code was successfully being built in local.

MoveIt!

During the implementation of the pick routine the code (completely working in local) didn't work on VLAB failing to plan every type of movement even when the robot didn't have any obstacles around, the code started working after using `.setPoseReferenceFrame("base_footprint");`.

To ensure that the Pick subroutine performs linear movements after approaching the object from above, we used the `.computeCartesianPath()` function from *MoveIt!*, but while it works perfectly on local, it continues to fail planning on VLab saying that the amount of linear path is below 50%, the same problem was experienced by some colleagues in another group. This part of

the code is currently commented to ensure that the code works on VLab, but classical planning using `.plan()` does not have the same guarantees that the movement will be linear.

References

[1] ROSWiki. Writing a callback based simple action client. `https://wiki.ros.org/actionlib_tutorials/Tutorials/Writing%20a%20Callback%20Based%20Simple%20Action%20Client`.