CS-7630 Autonomous Robotics

Homework – Behavioral Robotic

Support: cedric.pradalier@georgiatech-metz.fr
Office hours: 10:00 to 18:00.

This homework assumes that you have completed the teleoperation and collision avoidance homework.

Submit the two ros packages (tar.gz or zip) by email by next Monday, 10:00 to cedric.pradalier@georgiatech-metz.fr (don't forget to mention you group name)

Step 0: The Task System

Scene: rosControlBehaviours.ttt

One of the objectives of this homework is to make you experiment with a modular task composition system. This is slightly overkill for the tasks below, but it will be very helpful in future homeworks.

In the resource section of Piazza, you will find a small pdf document summarizing how to use the task framework developed by our robotic team. For your convenience, we prepared the task management system for the bubbleRob It implements a Goto(x,y) functionality as well as other tasks.

Setup you environment by copying the following directories into your catkin workspace and running catkin_make (from `/cs-share/pradalier/vrep_ros_ws/src`):

- `vrep_ros_teleop`
- `floor_nav`
- `ros_task_manager`

Start the simulation in V-Rep and launch the launch_vrep.launch file from the floor_nav package. This launch file might need to be modified to point at the right place for the teleop package. Assuming the launch files starts correctly, nothing should happen.

Then run the missions/test_goto.py script from the floor_nav package. The robot should drive safely around the scene and come back to its starting point.

If you reach this point, your setup is correct and you can start working on the project objectives.

In the following, you will need to create different behaviors (or tasks). In each case, a task to use as a starting point is given. For each new task, there are 3 files that need to be copy-pasted: the parameter description, in the cfg directory, the task header (.h) file and the task implementation (.cpp) file in the tasks directory. You then need to modify the CmakeLists.txt (a bit more copy-pasting), and the 3 files to replace any occurrence of the old task name with the new task name.

Step 1: Wandering Behaviour

The goal of this part is to tune a wandering behaviour: the robot should move forward as long as there are no obstacles, turn away from the obstacles and keep moving forever. TaskWander un floor_nav already implement this behaviour.

To follow the spirit of using simple sensors, the task is actually building first a simplified representation of the laser scan, by clustering laser points in angular sector with cfg.*angular_range* width (default: 30 deg), and computing the minimum range in each sector. After that, the *sensor_map* variable contains the simulated simple sensor reading: *sensor_map[0]* represents the sector [-30,0[, *sensor_map[1]* represents [0, 30[, etc.

Test it by creating a trivial mission that run only the Wander task.

By default, the code is designed so that there is no front looking sensor. By using *round* instead of *ceil,* (selectable with cfg.front_sector) it is possible to implement a similar clustering that simulates a front looking sensor (substracting *angular_range/2* would also work).

Modify the mission to instantiate the front looking sensor. What is the consequence?

Step 2: People Finder

The subsumption mechanism allows combining behaviors by making them replace each other's output (among other things). In the context of the task system, we will actually build something a little bit more deterministic.

The task system allows launching tasks in the background that can interrupt the main task. Check how it is done in the mission_cond.py in the task_manager_turtlesim package.

For this objective, you need to create two additional tasks:

- TaskWaitForFace which will wait until a face is detected. Use TaskWaitForROI as a starting point.

- TaskStareAtFace which will look a for a few second at a face, then turn away from it. Use TaskSetHeading as a starting point.

Both of these tasks will need to subscribe to your face publisher. You can add the subscriber in the task environment or as a member of each task. You can then create and initialise the subscriber in the initialize function of a task.

Based on these three tasks, write a simple mission that wanders aimlessly until a face is detected and then stares at the face, turn away and resume the wandering. Note: you can use the task SetHeading with the parameter “relative” to turn away.

Step 3: GoTo

The floor_nav package also implement a GoTo(X,Y) task that instruct the bubblerob to go to a given position, without orientation constraint. Create a new task, called GoToPose(X,Y,theta) that will go to a position with an orientation constraint. Add a boolean parameter to select between the dumb and the smart control method. To implement this task, start from TaskGoTo.

Modify the test_goto.py mission to use your GoToPose, and make sure you set challenging orientation targets.

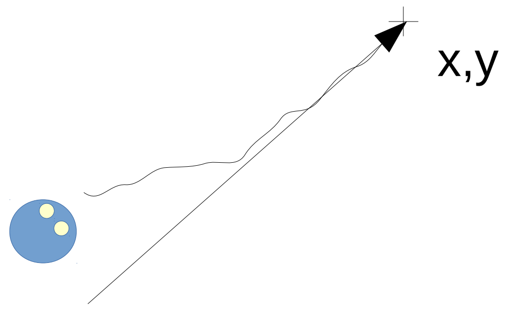
Step 4: TurtleBots

Try your mission on the turtlebot (the task environment subscribes to the simulated laser scan). You may need to change a bit the parameters, but not to write any code. Use the launch_turtlebot.launch in floor_nav.

Step 5: GoToDock

This last point is not due within this week, but we will need it later. If you have time this week, I would suggest implementing it now.

To dock on its charging station, the Turtlebot needs to approach a given position on a given line. We will use our line following control law in a new task called GoToDock(x,y,theta) that goes until the x,y position along a line supported by the (cos(theta),sin(theta)) vector.



Implement the GoToDock(x,y,theta) task starting from GoTo(x,y) and write a mission to test it in V-Rep.