Robotic Systems Programming - BOT
Practical Session 4 (TP-4)
Robot Control

Panagiotis PAPADAKIS

1 Introduction

The objective of TP-4 is to develop a *reactive* and a *planning* process for the TurtleBot2. The reactive process will be responsible for blocking/stopping the robot motors upon the activation of certain trigger events. The planning process will be responsible for computing (but not executing) a shortest 2D path from the current position of the robot to a given 2D position inside a 2D occupancy map of the environment.

1.1 Start-up

Connect to `eduroam`. Switch on the robot of your team and then switch on the mounted NUC computer. Wait around half a minute for it to boot and then you should be able to remotely connect via ssh to your TurtleBot2 via:

```
% ssh turtlebot@<robot>.priv.enst-bretagne.fr
```

where `turtlebot` is the username of the account of your robot and `<robot>` is the name written on the sticker of your robot (password is `613eluTUR`). This should allow you to run ROS within the NUC computer of the turtlebot. Run a `roscore` process in your turtlebot and in another terminal local to your machine execute:

```
% export ROS_MASTER_URI=http://<robot>:11311
```

to specify that the master ROS process is running on the robot[1] and:

```
% export ROS_IP=<your_ip>
```

to specify the IP of your own machine, inside the ROS ecosystem[2]. From this

[1] Remember that for every new terminal session you will have to set again the ROS_MASTER_URI

[2] Remember that for every new terminal session you will have to set again the ROS_IP

point forward, you should be able to run a ROS node either in your turtlebot or in your VM machine and all these ROS nodes should be part of the same ROS ecosystem. However, ROS nodes that are responsible for starting up robot sensors (such as the RGB-D camera), should only be run on a terminal of the robot and NOT from your machine[3].

Mapping using the real robot By default, when the robot computer starts there is nothing that is launched automatically with respect to ROS. Through a terminal session connected to your Turtlebot via `ssh`, go ahead and start-up TurtleBot2 drivers for sensors and actuators, using the following command:

```
% roslaunch turtlebot_bringup minimal.launch
```

If bring-up was successful, a *ascending* tune will be heard from the robot.

Then on another terminal where you again connect remotely to the robot via `ssh`, start the ROS nodes that will use the RGB-D sensor (camera of the robot) to create a map of the environment, using:

```
% roslaunch turtlebot_navigation gmapping_demo.launch
```

Mapping in the case that you work on a simulated robot If you have to work in simulation, all the previous steps are not necessary and you can simply launch the gazebo simulation:

```
% roslaunch turtlebot_gazebo turtlebot_world.launch
```

and then launch the mapping demo via:

```
% roslaunch turtlebot_gazebo gmapping_demo.launch
```

Moving the robot to map its environment Then, start a ROS package that will be useful for you to move the robot. You can use your own ROS node that you built in the previous TPs or the ROS package `turtlebot_teleop`. Prefer to run this on a terminal remotely connected on the robot. Also, take care that the control commands are sent to the appropriate ROS topic Attention: only send low-speeds to your robot ($. < 0.2m/s$), to avoid crashes and accidents.

Visualization Finally, start the ROS rviz visualization software (from a terminal of your own machine, NOT remotely to the robot) to see the TurtleBot inside the map. Add to rviz the following new markers:

- Map

[3]IMPORTANT: Notice that there are multiple independent ROS ecosystems, one per robot-team, that coexist in the same network (eduroam) without conflict, because the different roscores run on different IPs

- TF
- RobotModel
- IMU

Verify that you can see in rviz the respective visualized data, namely, a 2D occupancy map of the environment, the coordinate frame transformation tree (TF), a 3D model of the robot and IMU data. Save the rviz configuration file from the menu of rviz so that rviz will remember this configuration whenever it is launched.

After having launched `roslaunch turtlebot_navigation gmapping_demo.launch` (or the equivalent command in simulation), teleoperate the robot in a constrained area of the classroom, in order to create a 2D map of the environment. You should be able to see it within rviz using a *map* visualization marker. Save that map for later use using the ROS package map_server:

```
% rosrun map_server map_saver
```

which saves the current occupancy map into an image file .pgm and a corresponding meta-data file of type .yaml[4]. The next figure shows an example of a map.

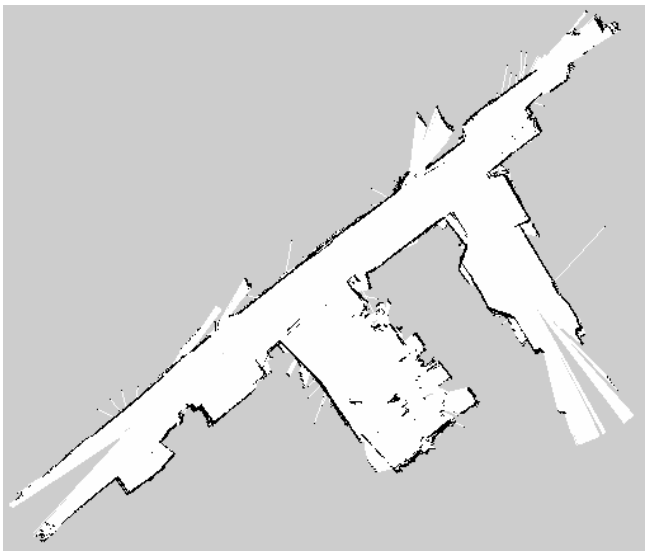

Figure 1: Example 2D occupancy map created by the turtlebot2

[4]NOTE: you can later load that saved map by publishing the corresponding message to ROS topic `/map` via `% rosrun map_server map_server map.yaml`

2 Exercises

2.1 ROS workspace creation

Create a ROS workspace that you will use to create and build your ROS packages (recall instructions of TP1, section 4.3, `https://moodle.imt-atlantique.fr/pluginfile.php/32285/mod_folder/content/0/TP1.pdf`).

2.2 Reactive process

Create a new ROS package called `tp4_ros_package`. This package should contain two ROS nodes (i.e. two python scripts), the first for your *reactive* process and the second for the *planning* process.

For the reactive process, you should develop a ROS node that listens to the IMU (Inertia Measurement Unit) sensor ROS topic:

```
/mobile_base/sensors/imu_data
```

which can be accessed after having launched the Turtlebot2 in simulation, via:

```
% roslaunch turtlebot_gazebo turtlebot_world.launch
```

A ROS message that is published in that topic is of type `sensor_msgs/Imu` and contains the orientation of the resultant force that is exerted on the turtlebot at a given moment. If the robot is static, this force corresponds to the force exerted by the ground to the robot as a reaction to the gravity force that is exerted on the robot to the ground (see Figure 2). If the robot is moving, the force of the motors that move the robot is further added. If the robot crashes or hits an obstacle, then an additional force is exerted to the robot from the collision contact.

As a result, when an obstacle is hit, a drastic change happens in the direction of the resultant force measured by the IMU. Your ROS node should detect this change and immediately disable the motors of the robot when such situation occurs.

IMPORTANT: In this way, the robot should be able to detect crashes **from any possible direction, either front, side, or back** in case it is moving backwards.

You should look into the documentation of the ROS messages that are communicated in the `/mobile_base/sensors/imu_data` topic to be able to retrieve the orientation and similarly for the ROS topic `/mobile_base/commands/motor_power` which is where your node should send the ROS message to disable the motors.

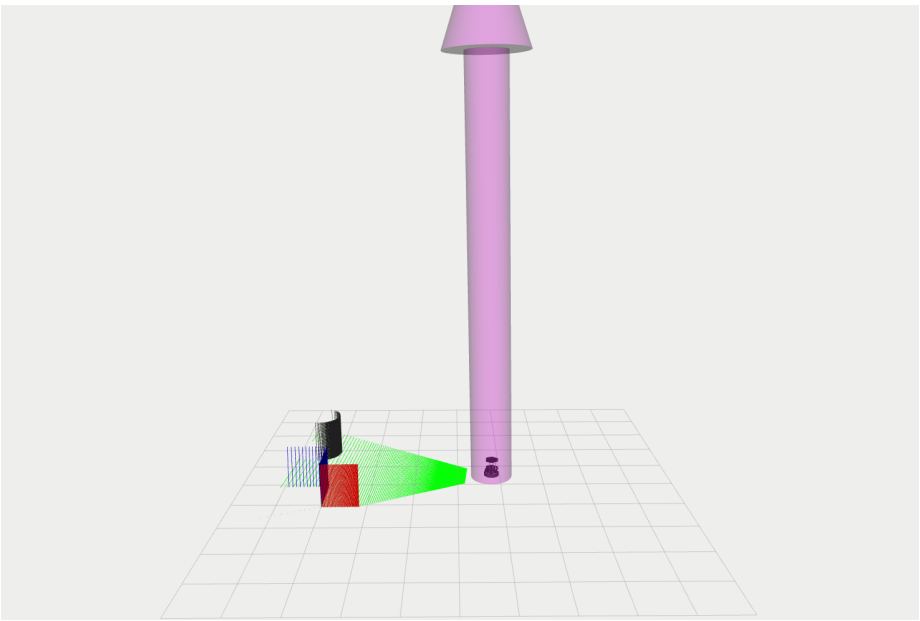

Figure 2: Force exerted to a static robot by the ground (viewed from ROS rviz)

2.3 Planning process

You should develop a ROS node (inside the same ROS package created earlier) that will take as input:

- A map message of type `nav_msgs/OccupancyGrid` from the topic `/map`
- A start (i,j) position of the robot inside the 2D occupancy map
- A goal (k,l) position of the robot inside the 2D occupancy map

and perform an A* search to find a path inside the constructed map, from a 2D start position of the robot to a goal 2D position (see Figure 3) assuming that the Turtlebot is a point robot. Your code should allow to use a 2D state space that is either *4-connected* or *8-connected*, which should be determined by a ROS parameter read by your node. The heuristic function used to guide the search of A* should depend on whether the *4-connected* or the *8-connected* neigbhorood case is used.

You are free to choose how you want to visualize the computed path from the initial state to the desired state. Along with your code, you should provide a total number of 10 different pairs of (start, goal) combinations along with the computed path, in order to demonstrate the effectiveness of your code.

You are free to choose whether you want to use your own map files constructed in simulation, in reality or preconstructed maps from games or real environments from `https://www.movingai.com/benchmarks/grids.html`

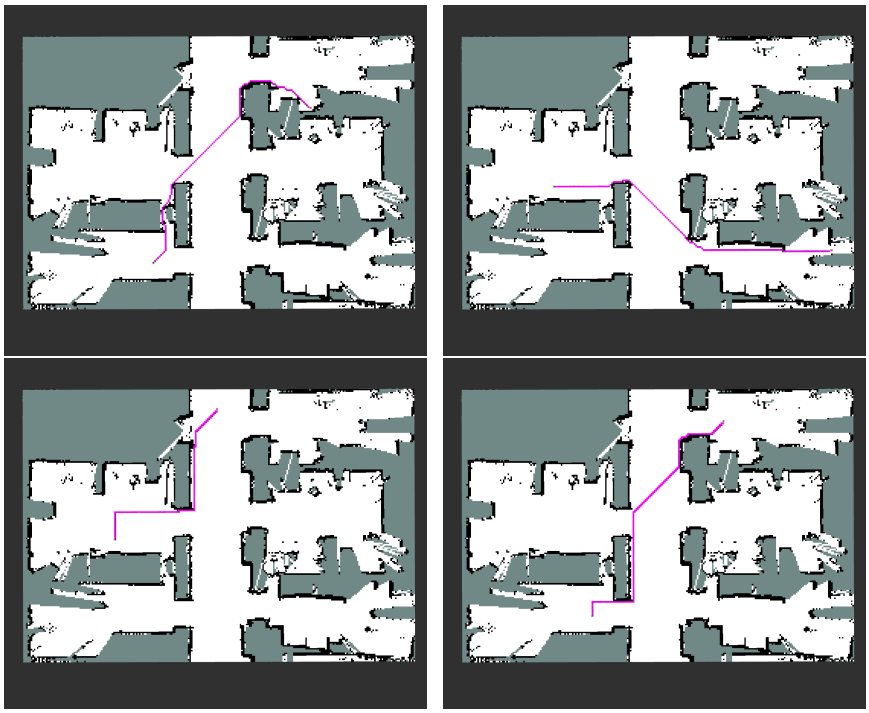

Figure 3: Paths computed with A* using 8-connected neighborhood (top row) and 4-connected neighborhood (bottom row)