Assignment #1 - Escape of Turtlebot

Due Date: check Ninova
This is an individual assignment, group work is not allowed!
Any type of cheating is strictly prohibited. Please, submit your own work!
You can ask your questions via e-mail (bicer21@itu.edu.tr) or message board in Ninova

Summary

Assignment: Turtlebot is trapped in a maze-like environment. As a fellow engineer of the robot, you have sent the coordinates of the exit point to it. Turtlebot will use a motion planning algorithm (Bug1 or Bug2) to reach the exit point. **Choose the suitable algorithm** for the maze (considering the elapsed time) and implement it to help our little friend.

Submission Type:**A single Python file** containing the source code of your controller. The file to be submitted is a1_answer.py. A skeleton for this file should be found in an archive supplied with the assignment description. It should be possible to compile the package by **placing the file into a ros2 workspace** as described in further and running the colcon build. **Do not forget to add your id**'s to your code as a comment in the specified section.

A 1-2 page report that addresses followings:

- What is the reason behind your choice of motion planning algorithm?
- What would have happened if you chose the other algorithm? (You can add supporting screenshots from the simulation)
- How did you implement the chosen algorithm? Explain what have you done to achieve each step of the algorithm in terms of implementation.

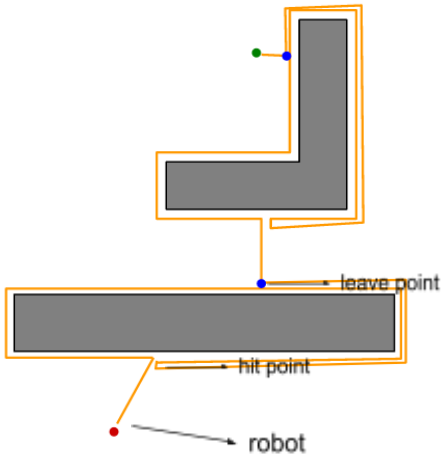
Set up your workspace

Before starting, make sure you have the necessary tools. For this assignment you will need the **Ubuntu 20.04** operating system, **ROS Foxy**. In order to use Ubuntu 20.04, it is recommended to install it standalone or alongside your usual operating system such as windows. You may also use virtual environments for this purpose although it is not recommended. You will need at least 20gb of free space. Warning: Make sure to get a backup for your documents before starting to install operating system, it is easy to make mistakes even if you are familiar with installing operating systems.
After installing Ubuntu (preferred) 20.04, the quick way to complete the rest of the installation is to use the script "**setup.sh**" from "**Class Files**". **Download the script** and **run it in your home directory**. Execute following commands subsequently and read the instructions in script.
Change mode to executable: chmod u+x setup.sh
Execute: ./setup.sh
You will see that our ros2 workspace folder is created automatically named as "ros2_ws" in home.
In such a case where you can not boot one of your operating system, you can repair it using boot-repair tool. You can use this tool by booting Ubuntu from usb. For details to use boot-repair, you can check the following link.
https://wiki.ubuntu-tr.net/index.php?title=Boot-Repair

Bug1 and Bug2

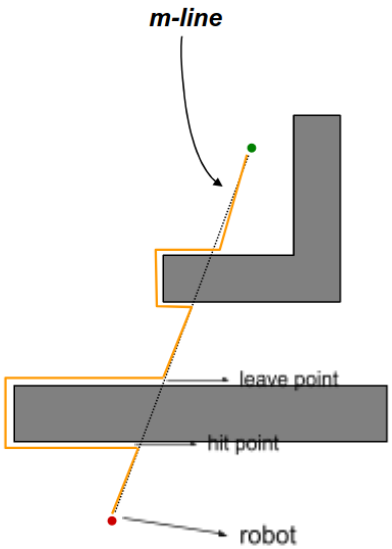
Special attribute of these algorithms is that they do not need global information (such as map information). Motion of the robot is planned according to the global goal and local sensing (e.g. laser scan data).

Bug1



1) **Head towards the goal** (green)
2) If there is an obstacle in the way; **turn left or right** and **follow the obstacle's circumference** (**and memorize how close you get to the goal**) until you come to the point where you encountered the obstacle (hit point) .
3) Return to the closest point near the obstacle (blue, leave point) and continue (step 1)

Bug2



1) **Head towards the goal** (green)
2) If there is an obstacle in the way; **turn left or right** and **follow the obstacle's circumference until you come across to the imaginary line** between starting point and the goal coordinates (m-line). It is important that robot should leave the wall **if and only if** the **leave point is closer to the goal than the hit point. Otherwise, continue following the obstacle.**
3) Leave the obstacle at that point and continue (step 1)

⚠️Regardless of which algorithm you choose, you should determine which side robot should turn (left or right) when it encounters with the obstacle beforehand and robot should turn to that side for every decision (e.g. robot should not be turning to right when it turned to left in previous obstacle without any reason).

Skeleton Code & Referee and Preparing Solution

"**a1_referee**" and "**a1_answer**" are two ros nodes in this assignment. Referee checks **whether robot has reached out to the goal spot and in how many seconds it did.** Periodically prints out the elapsed time and remaining distance to the goal spot. Source code for answer node **will contain your motion planning implementation to move the robot towards the goal as fast as possible**. A skeleton code is given for guidance.

There is a launch file ("a1.launch.py") that executes following:

- Gazebo
- Launch file of Robot State Publisher
- Turtlebot3 Cartographer Launcher (for building map while navigating)
- RViz
- Referee Node

To finalize the preparation of workspace:

- Move the downloaded "**assignment1**" folder to **"~/ros2_ws/src"** folder in home.
- Check dependencies: within the **"~/ros2_ws" execute "rosdep install -i --from-path src --rosdistro foxy -y"**
- Within the **"~/ros2_ws", execute "colcon build --symlink-install" command**
- Ignore bash warning in this terminal, it would disappear after you open new terminal.

Open up a new terminal and execute launch file by:
ros2 launch ~/ros2_ws/src/assignment1/launch/a1.launch.py

Gazebo and RViz will open up and you will see our maze-like world. **Goal is to reach (x:-1,y:-2)** coordinate as it is shown by black arrow in the Fig. 1. **As launch file is started up, you will see our referee node prints the distance between the robot and the goal, and elapsed seconds when robot starts its movement. Robot starts to measure elapsed time whenever robot starts the movement. When robot comes near to the goal point with 0.2 distance, referee accepts that robot achieved the goal and saves the elapsed time as time.txt (it would be saved in which location you executed the launch file).** In RViz, you can see the **map that cartographer builds, robot, TF2 frames, LaserScan points** (see Fig. 2). **Red marks are the scanned points**. You can also enable "Trajectories" within the cartographer on the left pane. This would mark the path that robot has taken.

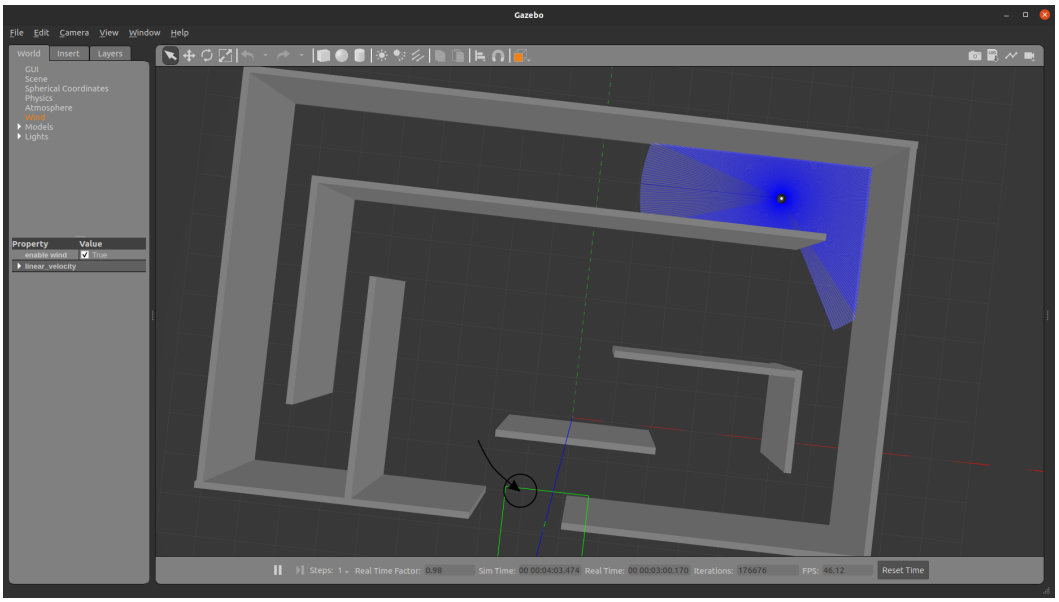


Fig 1. Gazebo simulator with our world. Here arrow represents the goal. You can see our little friend on top right also.

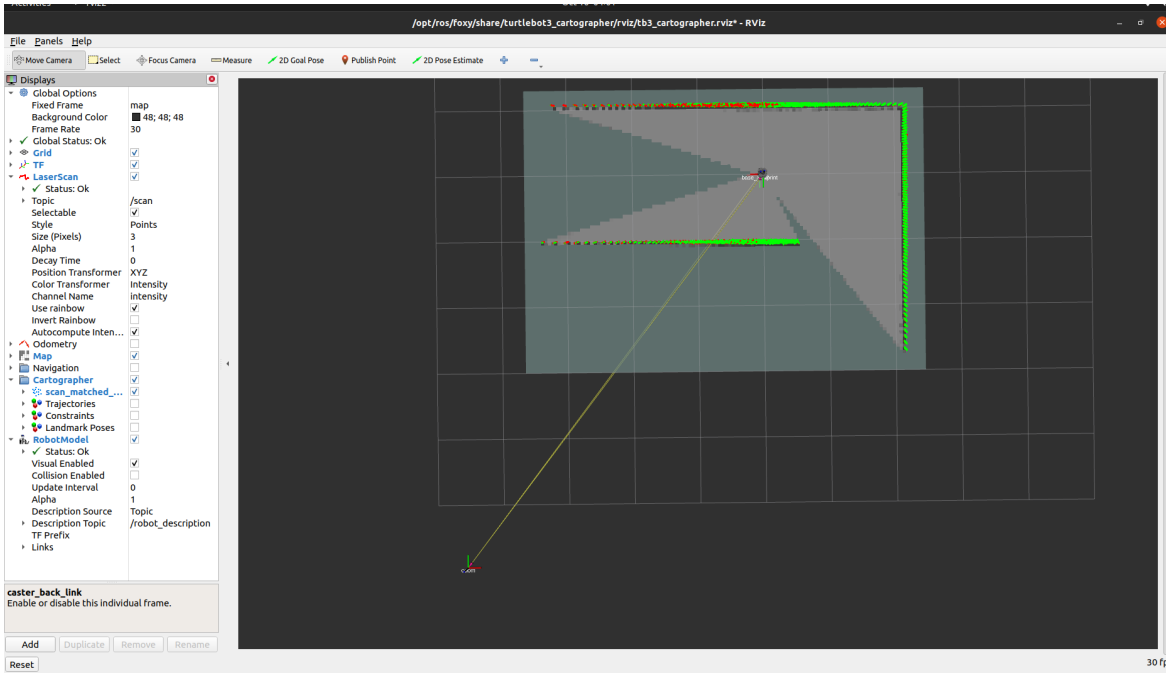


Fig. 2. RViz window after executing a1.launch.py.

Choose the suitable algorithm between Bug1 or Bug2 for this world, implement it within the "a1_answer" code with guidance of codes that are intentionally left in it to help you. See below header (Information regarding callbacks of topics).

You can move the robot by teleoperating to play with it, execute following command after the launch:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

You can observe the ROS Computation graph by (see the transmission of messages):

```
ros2 run rqt_graph rqt_graph
```

Marking Criteria

- Smooth movement as much as possible (no need to stop for every second you move, stop only you need to)
- Fast as much as possible, yet avoid making robot fall down:
    - If your real time factor (check bottom pane in Gazebo) is between 0.90-1, **acceptable solution is maximum 3 minutes and 20 seconds.** If real time factor is between 0.70-0.8, **maximum elapsed time should be 4 minutes and 20 seconds. Evaluation will be on a computer that achieves 0.90-1 real time factor. Do the math for any other real time factors. Elapsed time and real-time factor is inversely proportional. Surpassing maximum elapsed time would be penalized.**
- Make use of laser scan, beyond what skeleton does.
- Avoiding collision with objects (Hit point in Bug algorithm does not say that you should hit the obstacle obviously).
- Modular code: divide operations into functions to make code seem more elegant.
- Code with comments (good practice)
- Sufficient explanation for specified questions in Report document.

Class competition

The referee measures the time it takes robot to escape the maze. Referee will print elapsed seconds and distance to the goal every second to the terminal. As a class competition, students with least elapsed time and smooth movement for escaping the robot would get a prize.

Information regarding callbacks of topics

How to use scan data: Laser scan data will be available within the callback function that your program registers when it subscribes to the /scan topic by creating a subscription object with "create_subscription" function. A LaserScan message is published through this topic, and detail about the data structure of this message can be found in:
https://docs.ros2.org/latest/api/sensor_msgs/msg/LaserScan.html

This message basically tells us how far are the objects that surrounding our robot via 360 laser sensors on its body. In skeleton code, helper code is left to guide you how to manage this message.

How to navigate the robot: You can send Twist messages to the robot to say how much velocity should it have in linear or angular aspect. This message can be used for three dimensional objects. As we are only concerned with two dimensions, you will only need to set the **x component of the linear** velocity and the **z component of the angular velocity** (anyway, the robot is only a wheeled mobile robot so could not follow many possible 3D motions that would require flight for example). As with the laser scan, you can access the fields of the **geometry_msgs/Twist** message (https://docs.ros2.org/latest/api/geometry_msgs/msg/Twist.html), which will be "linear" and "angular", which themselves are messages of type **geometry_msgs/Vector3**. **Skeleton code includes helper code in scan callback for sending movement messages.** To see what fields are available in a Vector3 message, see the message definition:
https://docs.ros2.org/latest/api/geometry_msgs/msg/Vector3.html

TF2 and getting Robot's Position: In order to obtain the robot's current pose the skeleton code uses the TF package to obtain the transform between the robot's original pose (which also happens to be the map frame as long as the robot's odometry agrees with the localization subsystem) and its currently known pose according to the odometry. These poses are known to the TF transform manager as frames /odom and /base_footprint. In order to use TF well, you can examine carefully the output of the following commands to view what is going on:

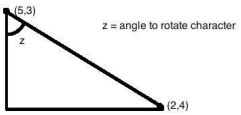
```
ros2 run tf2_tools view_frames

evince frames.pdf
```

Also:

```
ros2 run tf2_ros tf2_echo /base_footprint /odom
```

More information

atan2: It is used in the "euler_from_quaternion" method in skeleton code. The math.atan2() method returns the arc tangent of y/x, in radians. Where x and y are the coordinates of a point (x,y). The returned value is between PI and -PI.



atan2(diff x, diff y) = z
atan2(5-2, 3-4) = z
atan2(3, -1) = z

Map: The assignment itself does not require you to access the map. If you want to use it, the information in this section can help. The map is provided in an OccupancyGrid type message. Cartographer SLAM builds a map of the environment and simultaneously estimates the platform's 2D pose. The localization is based on a laser scan and the odometry of the robot.

Information about how to use the occupancy map message can be found here :

https://docs.ros2.org/latest/api/nav_msgs/msg/OccupancyGrid.html

In order to access map information you can use the following expressions:

Height in pixels: msg.info.height

Width in pixels: msg.info.width

Value (at X,Y): msg.data[X+ Y*msg.info.width]

Meaning of value:

0 = fully free space.

100 = fully occupied.

-1 = unknown.

Coordinates of cell 0,0 in /map frame:msg.info.origin

The size of each grid cell:msg.info.resolution

Obviously the robot does not have access to the complete map (the robot does not know the contents of the simulation/world only what its own sensors show) so this estimate can sometimes be wrong. If, in the skeleton code, you set the value of the (const bool) variable **chatty_map to true**, it will print an ASCII representation to terminal of the current map. This will quickly get untenable for bigger maps, but the **rviz session that is loaded with a1.launch will also show the current map as known/estimated by the robot.**