YZV 406E - Robotics
Assignment #2

Due: Dec, 1st 23:59

***Admiral Ackbar**: It's a trap!*

-Return of the Jedi (1983)

Introduction

For this assignment, you are expected to program a motion planning algorithm for the turtlebot. For any issues regarding the assignment, you can ask your questions on Ninova Message Board. Before writing your question, please check whether your question has been asked. You can also contact T.A. Erhan Biçer (**bicer21@itu.edu.tr**).

Implementation Notes

The implementation details below are included in the grading:

1. Please follow a consistent coding style (indentation, variable names, etc.) otherwise you will get **minus** points.
2. Add comments to your code to explain each function or section of code.
3. Modular code structure. Divide your code into effective functions such as "go to point", "wall following". You can include helper functions. **Your code should be understandable!**

Submission Notes

1. **Submit a 1 page pdf that addresses followings**: (1) Draw two worlds for scenarios that Bug1 beats Bug2 and Bug2 beats Bug1 respectively. Draw unique worlds, not the same worlds you observed in lecture notes. (2) Explain how your algorithm works step by step. You can mention functionalities of your functions. Do not explain how Bug 1 works, you should mention how you manage to achieve steps of Bug 1.
2. Your program should compile and run on ROS2 Foxy Fitzroy. You can (and should) code and test your program on Ubuntu 20.04. It is not recommended to test your code in different environment setups. Be sure that you have submitted all of your files.
3. Submissions are made through **only** the Ninova system and have a strict deadline. Assignments submitted after the deadline will not be accepted.
4. This is not a group assignment and getting involved in any kind of cheating is subject to disciplinary actions. Your assignment should not include any copy-paste material (from the Internet or from someone else's paper/thesis/project).

Implementation Details

Warning: Review the local motion planning notes before.

You are expected to implement a **Bug 1** motion planner for the turtlebot. Turtlebot will go to a specific goal point **(x=1,y=3)**. We will be using Gazebo for simulation, and rviz to see how turtlebot sees the world and prepare the map by using its sensors. Your simulation environment is seen in Fig. 1. In RViz, you can see the map that cartographer builds, robot, TF2 frames, trajectory of the robot, LaserScan points. Red marks are the scanned points. You can examine the frames of the robot, odom frame and map frame in RViz.

You can create functions and define variables according to your needs. Examine the `motion.py` code and understand which parts need to be modified. All dependencies that you are going to need are provided inside "package.xml" and "CMakeList.txt" file. You can use standard Python packages or ros packages if you needed additionally. Note that those packages cannot include ready-to-use Bug algorithm implementation. You need to implement Bug 1 algorithm from scratch.

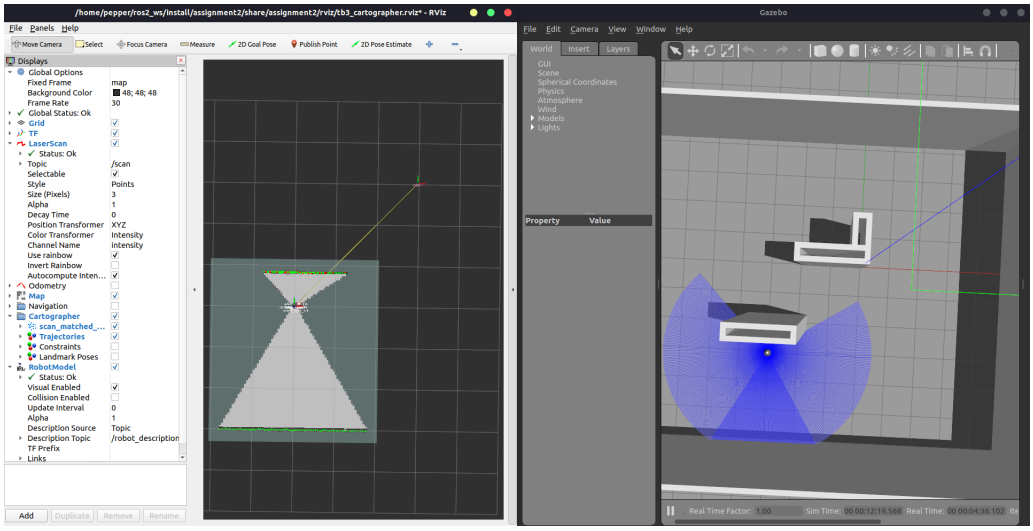

Figure 1: Gazebo and RViz.

In this assignment, you are expected to develop a motion planner algorithm, specifically Bug 1 algorithm, to move the robot to the goal point (Fig. 2).

After implementing the standard Bug 1, you may improve the algorithm by optimizing the taken path to make the robot motion more efficient in terms of length of taken path (**This is optional.**). Even if you try to optimize the taken path, you cannot change the basic structure of the algorithm. Your algorithm should follow the 3 basic step nevertheless: (1) head towards the goal, (2) circumnavigate the obstacle (3) return to the closest point to the goal.

If you examine the "tracker.py" file shared with you, you can see that: (1) it measures the **distance between the robot and the goal point**; (2) it measures the **elapsed time after motion has started**; (3) it stops the robot when the goal is achieved. **If robot is near to the goal point by 0.2 or less units, goal is achieved.** It logs these information on terminal. After achieving the goal, **elapsed time will also be saved into a text file called time.txt** which can be found at the current directory of the terminal where launch file is executed.

After checking the dependencies with rosdep and building the package, you can run the assignment by writing "`ros2 launch assignment2 a2.launch.py`" in the terminal. You can utilize the "`colcon build --symlink-install`" option to make coding and testing easier. You can run your solution with:
"`ros2 run assignment2 motion.py`".

You can explore the world with teleop as we did in turtlesim:
`ros2 run turtlebot3_teleop teleop_keyboard`
You can observe the ROS Computation graph by (see the transmission of messages): `rqt_graph`

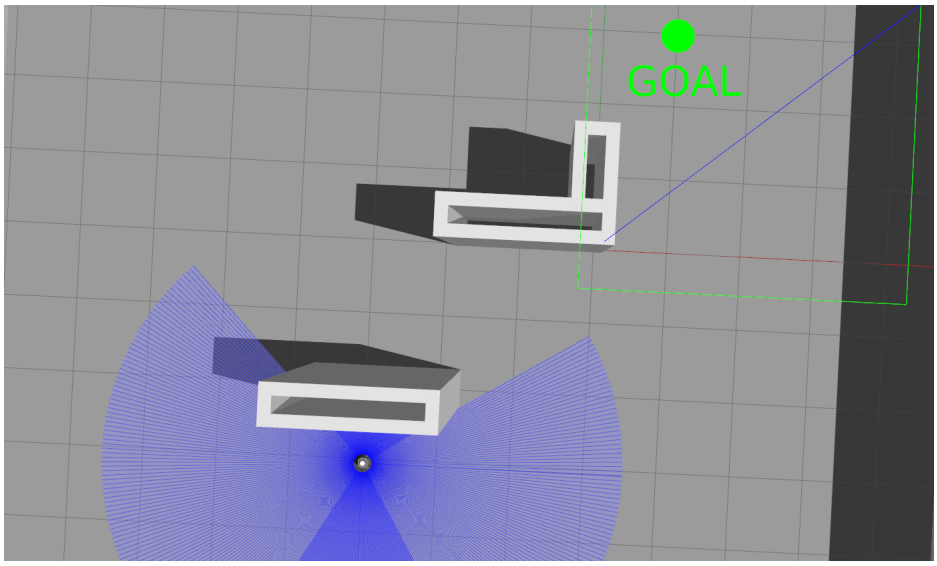

Figure 2: Goal point shown in gazebo.

Marking Criteria

1. Make use of laser scan, beyond what skeleton does
2. Avoiding collision with objects (Hit point in Bug algorithm does not say that you should hit the obstacle obviously)
3. Fast motion as much as possible, yet avoid making robot stumble and fall:
   (a) 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 can be roughly 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.
4. Smooth movement as much as possible (no need to stop for every second you move, stop only you need to)
5. Modular code: divide operations into functions to make code seem more elegant.
6. Code with comments (good practice)
7. Sufficient explanation for specified questions in Report document.
8. Correct implementation of Bug 1 algorithm.

Information regarding 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 ros documents. 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**. As we are only concerned with two dimensional movement, **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). You can access the fields of the geometry_msgs/Twist message (ros documents), 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

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 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 represents angle between PI and -PI. This function can be useful in your implementation. See Fig. 3.

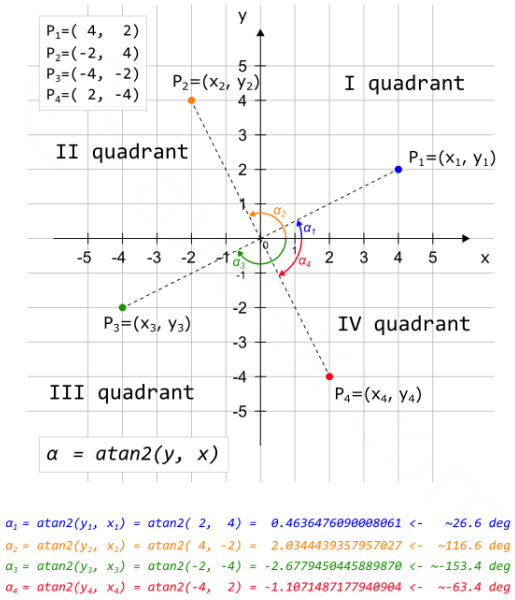

Figure 3: atan2 function

Examining map information The assignment itself does not require you to access the map. If you want to examine it, information at below would be useful.

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

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

1. Height in pixels: msg.info.height
2. Width in pixels: msg.info.width
3. Value (at X,Y): msg.data[X+ Y*msg.info.width] Meaning of values:
   0 = fully free space.
   100 = fully occupied.
   -1 = unknown.
4. Coordinates of cell 0,0 in /map frame: msg.info.origin

5. 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 a2.launch will also show the current map as known/estimated by the robot.