Assignment #1 - TurtleBot the Explorer

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 instructed to explore the map you have created. As a fellow engineer of the robot, you need to make this possible by publishing movements that make the robot navigate around the world you have created. You will create the world using Gazebo. The world **will include rooms** where robot needs to explore. The robot **can explore rooms and exit the rooms** using the laser information. You need to write a custom ROS2 node that explores the area. You will be utilizing **laser sensor** values of the TurtleBot (/scan topic) to **avoid obstacles** while the robot wanders around. The robot **should continuously navigate** around the world. Your solution may be tested on another world, so avoid implementing a navigation node that is developed specifically for your world. Your solution will be tested for 2 minutes at maximum.

Submission Type: **A ZIP file that contains turtle_explorer.py, "/models" (folder), "/worlds" (folder), "report.pdf", "my_map.png", "my_map.yaml".** Source code of your controller to be submitted is called **turtle_explorer.py**. A skeleton for the file can 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 single page report that addresses followings:

- 1-2 sentences that explain the world you have created and the robot's role in the world.
- Explain how you have managed to explore the map without hitting an obstacle.
- Report the explored area with percentage (include the extracted map).
- Explain challenges and how you have managed to overcome them.
- **Include the robot's initial pose arguments in your report (You may write the whole launch command).**

Skeleton Code, Referee and Preparing Solution

Tracker Node checks followings:

- **Percentage of the explored area of the map**
- **Elapsed time in seconds**

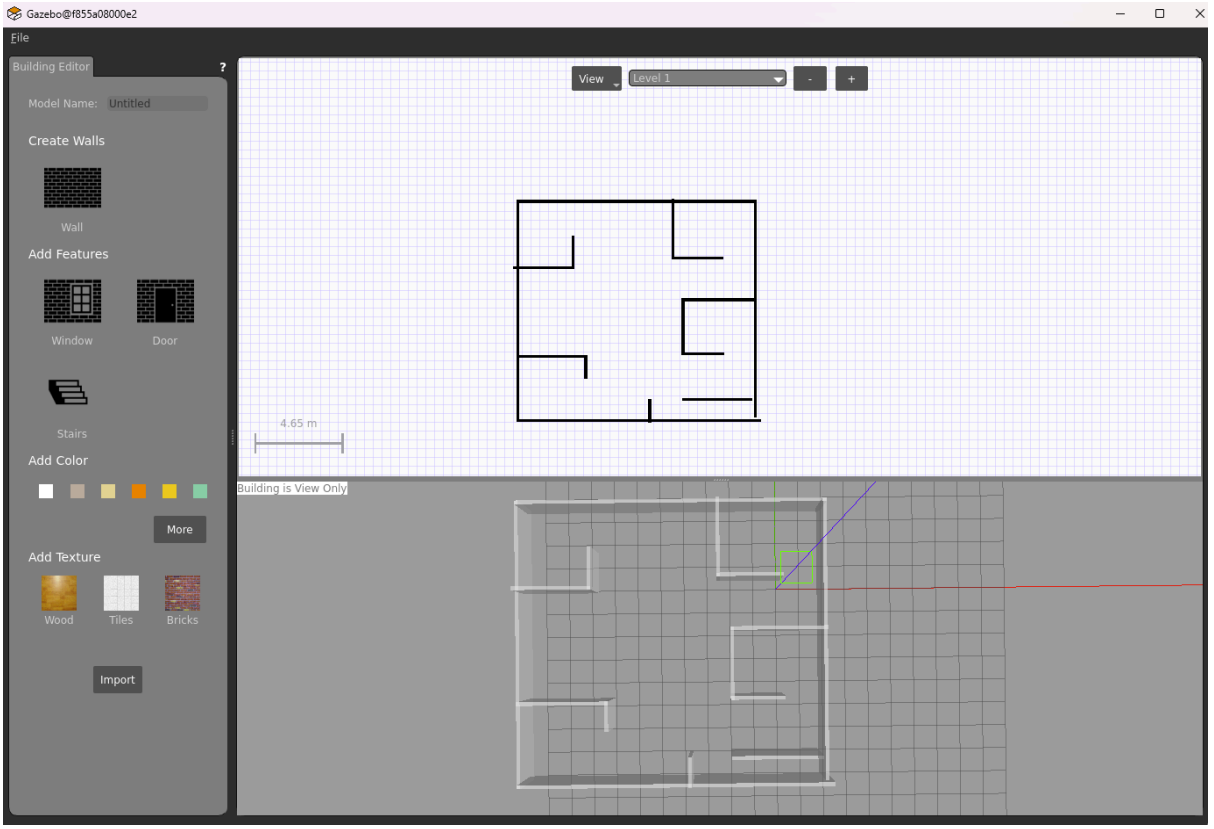
Tracker would log the information above in the terminal. **The robot should explore the world and avoid obstacles. Tracker would save the extracted map after 120 seconds as .png file under the directory where the launch file is executed.**

Creating the map with Gazebo

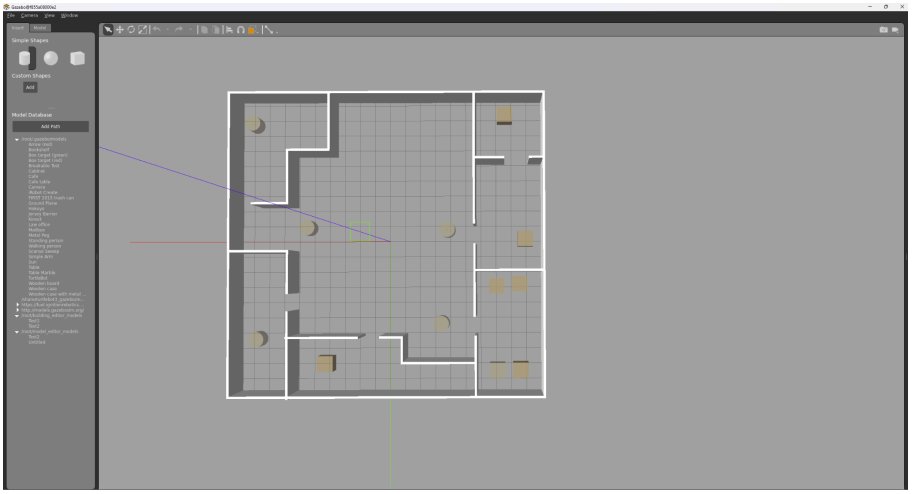
In this assignment, the map where robot wanders will be designed by you! The designed world should be a location within ITU Campus. Design your world according to that location (e.g. Faculty Entrance Floor) You need to follow below steps to create the map with gazebo:

1. Open gazebo with the command: `"gazebo"`
2. Select "Building Editor" from the "Edit" menu.

3. A window like below should appear:

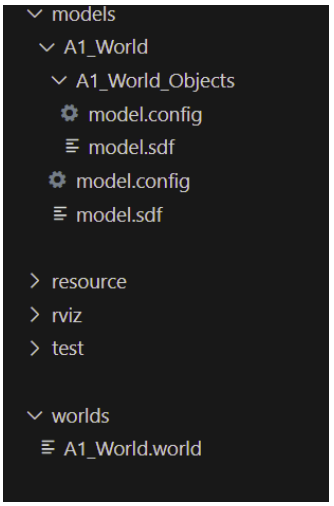


4. You can select the Wall from the left pane and place it onto the map. Placements are done in the top pane where the background is fully white. You can guide yourself about where to place by checking the real location in the below pane. Ensure that your environment is enclosed with walls so that robot cannot escape the world. You can add texture to the buildings by selecting the texture and placing onto the structures in your world. There should be at least **4 room-like wall separations. Map should be at least 10x12 gazebo grids (below pane).** *Do not make huge worlds as it would take too much time to explore the world (14x14 at maximum)!*
5. After you are done with the world, you can save the world model by selecting “Save As” from the File menu. **Make the model folder name A1_World**. After saving (ensure that file is seamlessly saved by checking the file), you can leave the building editor by “Exit Building Editor” option from File menu. Now, you have the world, **you need to add also objects which will be encountered by the robot to create a more challenging environment. Select “Model Editor” from the “Edit” menu.** You can **at least include several simple objects (3-4)** like the image below from the Model Editor.



After you add an object, you can change the color of it by double clicking. Change the Ambient RGB values (between 0-1). Keep in mind that if there are numerous objects, FPS will be much lower. Save the model file by selecting “Save As” from the File menu.

6. After you are done, navigate to the File menu and select “Save World As” from File and save the world file. **Save the world file as A1_World.world.**
7. Place the .world file in the worlds folder in the assignment folder. Place the model files in the models folder. Here “A1_World_Objects” is the model folder for the obstacles that you have added. An example of world and model structure in package folder:



Run the launch file

1. ros2 launch a1 a1_launch.py
2. Run the solution code: ros2 run a1 **turtle_explorer**

Developing the explorer node

Source code for answer node **will contain your implementation to move the robot to explore the world as much as possible around 2 minutes**. The robot will **make use of the laser sensor** values coming from /scan topic to avoid obstacles while exploring the world. The robot may enter the room, it can leave the room, by sensing the environment, and resume its exploration of the world. A skeleton code is given for guidance. Map is also stored in variables (see Map in information section below).

There is a launch file ("a1_launch.py") that executes following:

- Gazebo
- MapServer for publishing map
- AMCL for localizing robot in the given map
- Nav2 Lifecycle to configure and launch MapServer and AMCL in coordination.
- Launch file of Robot State Publisher
- RViz
- Tracker Node

To finalize the preparation of workspace:

- Move the downloaded "**a1**" folder to **"~/ros2_ws/src"** folder in home.
- Check dependencies: within the **"~/ros2_ws" execute "rosdep install -i --from-path src --rosdistro humble -y"**
- **Move the world and model files into corresponding folders as aforementioned above.**
- Within the **"~/ros2_ws", execute "colcon build" command**

Open up a new terminal and execute launch file by:

- `ros2 launch a1 a1_launch.py x_pose:=-5 y_pose:=-3 world:=A1_World.world`

Then, skeleton code can be executed by:

- `ros2 run a1 turtle_explorer`

Gazebo and RViz (see Fig.1) will open up and you will see our world after executing the launch file. **Elapsed time and explored area are being measured after the robot starts the movement.** In RViz, you can see the **map that robot currently has knowledge of, robot, TF2 frames, LaserScan points. Red marks are the scanned points. RViz visualizes the robot's vision. As the robot navigates, the map gets updated,** so RViz's visualization becomes more similar to the actual world. RViz gets the map information through /map topic which is continuously updated by "Cartographer" according to the environment sensing due to laser sensors.

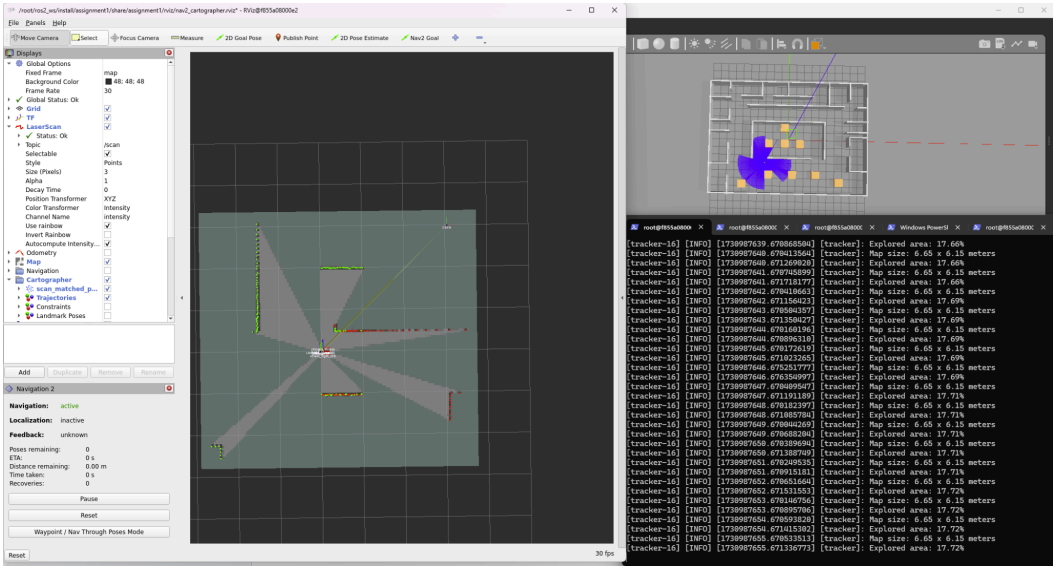


Fig 1. Gazebo simulator with our world. You can see our little friend TurtleBot on bottom left.

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
```

If you are curious about advanced navigation: *You can try out Nav2 path planning and navigation by selecting Nav2 Goal at the upper pane of the Rviz and place in the world, then you will see the planned trajectory and navigation of the robot! Nav2 is a navigation stack used in ROS2, which is out of scope of this assignment.*

Save the map by following command:

```
ros2 run nav2_map_server map_saver_cli -f my_map --fmt png
```

Marking Criteria

- Creation of a **unique** world
- Publishing movements that navigates the robot
- Exploration of the world through robot navigation
- Obstacle/Wall **collision avoidance**
- **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.

Hints:

- Using teleoperation to get the sense of robot navigation.
- Implementation of a controller like Proportional (P) control or another advanced controller.
- You may make use of /map topic for intelligent exploration.

Information regarding messages, topics and so on

You may check docs.ros2.org for more than below.

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.

This message basically tells us how far are the objects that surround 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 it should 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**. As with the laser scan, you can access the fields of the **geometry_msgs/Twist** message, 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 (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

Map: If you want to use it, the information in this section can help. The map is provided in an array with occupancy_grid variable. Provided map is updated using the robot's laser sensors. Since the robot does not have the map information beforehand, the map will be updated as the robot navigates and senses the environment using laser sensors.

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

- Value (at X,Y): `occupancy_grid[X+ Y*map_width]`
  - Meaning of value:
    - 0 = fully free space.
    - 100 = fully occupied.
    - -1 = unknown (not yet explored).
- Coordinates of cell 0,0 in /map frame (bottom left corner of the map):`map_origin`
- The size of each grid cell:`map_resolution`

For values between 0-100, by default, values greater than 0.65 are considered as completely occupied, values less than 0.25 are considered as completely free space.

If, in the skeleton code, you set the value of the (const bool) variable **chatty_map to true**, it will print the map as a list. Also, **the rviz session shows the map.**

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. **It is utilized to find the angle to face towards a specified point!**

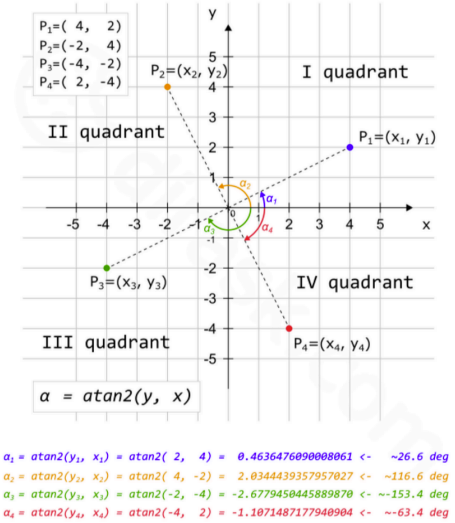


Fig. 2. atan2 function