Assignment #2 - Waypoint Follower DeliverBot

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 make a delivery. It has to take the package (in our case, an imaginary box) from the last waypoint of 1st route and deliver it to the last waypoint of the 2nd route. As a fellow engineer of the robot, you need to make this possible by publishing movements that make the robot reach to given waypoints. However, waypoints are based on an older version of the map. **Therefore it does not take the obstacle along the 2nd route into account.** You need to pass this object and deliver the package by moving to last waypoint of the 2nd route. **Publish movements that can reach to last waypoint without colliding with the object. You may require to use robot's range sensors or the map (OccupancyGrid). For other waypoints, you might not use range sensors if you do not want to.**

Submission Type:**A single Python file** containing the source code of your controller. The file to be submitted is waypoint_follower.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 single page report that addresses followings:

- Explain how you tracked the waypoints and the reason behind it.
- Explain how did you overcome the obstacle along the route 2.
- Mention extra things, if any, that you add to help the waypoint following of the robot.

Skeleton Code & Referee and Preparing Solution

"**a2_referee**" and "**waypoint_follower**" are two ros nodes in this assignment. Referee checks followings:

- **Has robot reached to the next waypoint?**
- **Does robot have the desired yaw (orientation)? (Robot should be facing the same direction as the arrow shows)**
- **How many seconds has passed for robot to reach to the next waypoint?**
- **How many seconds has passed for robot to reach to the final waypoint of the 2nd route?**

Referee would print the answers for these questions in terminal and also in text files as **wpReport.txt and timeReport.txt**. You can track your performance with these reports. wpReport includes waypoints that your robot reaches and how much time robot got late to that point as "WP0+,-2.30566668510437". This says that first waypoint is achieved and the yaw of the robot is

correct (i.e if there is no + sign, you only achieved that waypoint linearly). You got early to that waypoint by ~2.3 seconds as there is a "-" sign (i.e there would not be "-" if you reach to that waypoint late.). **It should be noted that your solution might be fast enough yet results in lateness for the waypoints. Required time to reach those waypoints are prepared in my computer which its performance may differ from yours. This report is given for you to track yourself and improve your solution iteratively.** timeReport will contain elapsed time and total lateness if all waypoints are finished.

Source code for answer node **will contain your implementation to move the robot towards the end of the route 2 by following waypoints as fast as possible**. A skeleton code is given for guidance. **Skeleton code already stores the route 1, route 2 and next waypoint in route1, route2 and waypoint variables of the node. Map is also stored in variables (see Map in information section below).**

There is a launch file ("nav_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 (map should be already included)
- Referee Node

To finalize the preparation of workspace:

- Move the downloaded "**a2**" 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**
- If encountered: ignore bash warning in this terminal, it would disappear after you open new terminal.

Open up a new terminal and execute launch file by:

1. cd ~/ros2_ws/src/a2/launch
2. ros2 launch nav_launch.py

Then, skeleton code can be executed by:

```
ros2 run a2 waypoint_follower
```

Gazebo (see Fig.1) and RViz will open up and you will see our world after executing launch file. **Goal is to reach to the last waypoint of the 2nd route. Robot starts to measure elapsed time whenever robot starts the movement.** In RViz, you can see the **map with the occupancy grid, robot, TF2 frames, routes, next waypoint and LaserScan points** (see Fig. 2). **Next waypoint will be appeared in green color for route 1 and yellow color for route 2. First route is colored as purple and second as red. Red marks are the scanned points.**

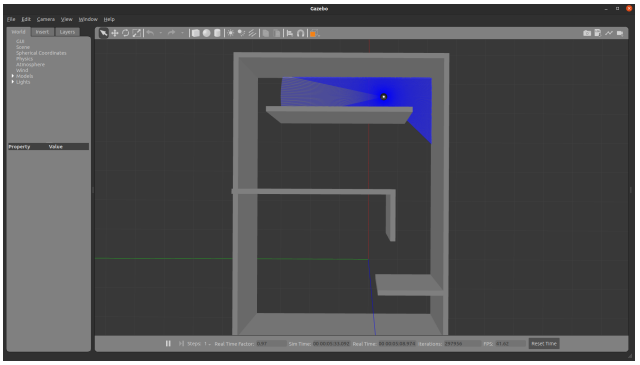


Fig 1. Gazebo simulator with our world. You can see our little friend on top right also.

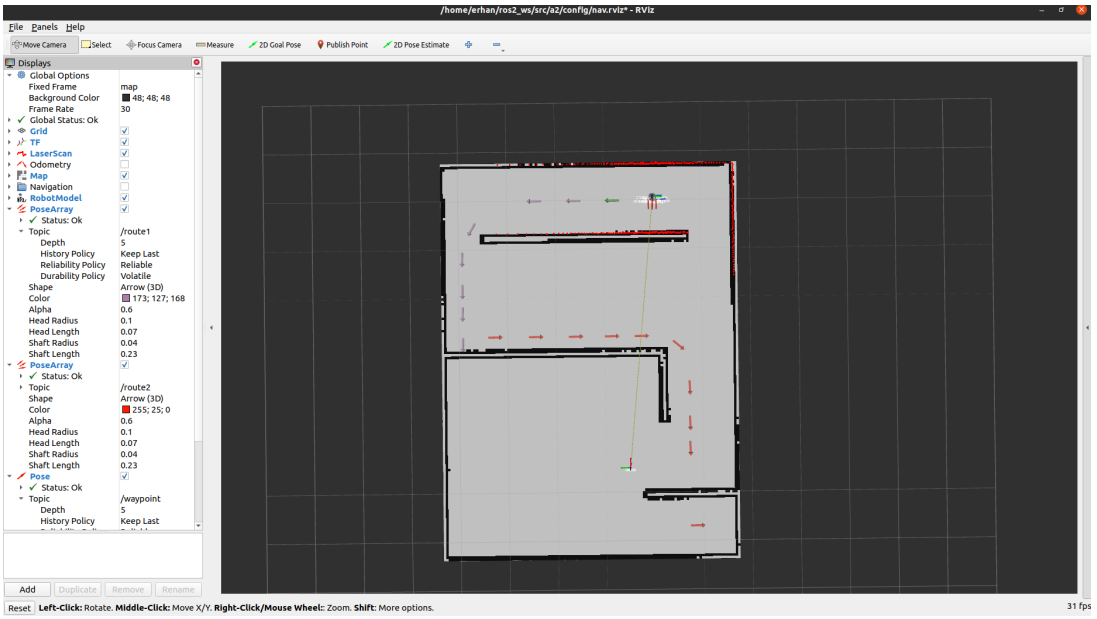


Fig. 2. RViz window after executing nav_launch.py.

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

- Publish smooth movements as much as possible (**do not stop for every second you move, stop only you need to**)
- Publishing movements that get the robot to the target position.
- Publishing movements that get the robot to the target position and orientation (desired yaw).
- Responding quickly to changes in waypoint (after you achieved a waypoint, new waypoint will be assigned.)
- Achieve routes 1 & 2 successfully by passing through waypoints.
- Publishing movements that can pass the obstacle and reach the last waypoint in route2.
- Fast as much as possible:
    - Elapsed time will be evaluated according to the best solution submitted. (**If you stop and run the robot again for each waypoint, your solution will be penalized. Remember that this is a delivery scenario, and your solution needs to be fast.**) You may discuss your time performance among yourselves.
- 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:

- Implementation of a controller like Proportional (P) control or another advanced controller.
- Figuring out and making use of the /route1 and /route2 topic also provided by the referee.
- Using teleoperation to get the sense of how waypoint following works.

Class competition
The referee measures the time it takes robot to deliver the package. Referee will print elapsed seconds to the terminal and timeReport.txt. As a class competition, students with least elapsed time and smooth movement for delivering the package would get a present.

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 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

PoseArray, PoseStamped, Pose and Related Topics:
Pose: Pose is a representation of pose in free space, composed of position and orientation. Position is represented with geometry_msgs/Point type (x,y,z) and orientation is in Quaternion format (x,y,z,w).
PoseStamped: PoseStamped is a Pose with reference coordinate frame and timestamp. It consists of a header additionally which includes timestamp information and associated frame (map frame in our case). We used timestamp to add expected approximate elapsed time for reaching the waypoint.
PoseArray: An array of poses with a header for global reference.
Related Topics: /route1 returns a PoseArray of waypoints along the route 1. **/route2** returns a PoseArray of waypoints along the route 2. **/waypoint** returns the next waypoint with the type of PoseStamped.

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
```

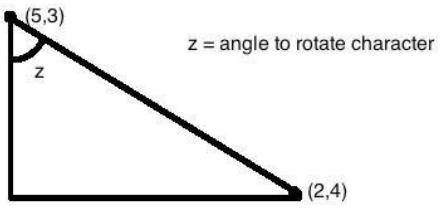
```
evince frames.pdf
```

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: If you want to use it, the information in this section can help. The map is provided in an array with occupancy_grid variable. Our map is 163 (map_height) x 122 (map_width) grids.

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.

Coordinates of cell 0,0 in /map frame:map_origin

The size of each grid cell:map_resolution

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, **rviz session that is loaded with nav_launch will also show the map. Different from the first assignment, we have access to the whole map since it is generated with slam (using Nav2) beforehand for this assignment.**